TAMPEREEN TEKNILLINEN YLIOPISTO

Anna-Liisa Mattila

3D-käyttöliittymäkomponenttikirjaston toteuttaminen web-tekniikoilla

Diplomityö

Tarkastajat: Tommi Mikkonen, Arto Salminen ja Jari-Pekka Voutilainen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan
tiedekuntaneuvoston kokouksessa
7. maaliskuuta 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
MATTILA, ANNA-LIISA: 3D-käyttöliittymäkomponenttikirjaston toteuttaminen web-tekniikoilla
Diplomityö, 54 sivua, 4 liitesivua
Lokakuu 2012
Pääaine: Ohjelmistotuotanto
Tarkastajat: Professori Tommi Mikkonen, assistentti Arto Salminen ja tutkija Jari-Pekka Voutilainen
Avainsanat: 3D-käyttöliittymät, WebGL, HTML5

Interaktiivisten kolmiulotteisten (3D) web-sovellusten kehittäminen on nykyään mahdollista, mutta ei kuitenkaan yksinkertaista. Interaktiivisten 3D-sovellusten toteutustekniikat ovat abstraktiotasoltaan matalalla verrattuna esimerkiksi interaktiivisten 2D-sovellusten vastaaviin. Graafisten 2D-käyttöliittymien kehittämistä varten on toteutettu lukuisia käyttöliittymäkirjastoja ja muita aputyökaluja, mutta interaktiiviset 3D-sovellukset toteutetaan vielä pitkälti 3D-moottorien ja mallintamisohjelmien tarjoamia palveluja käyttäen.

Tässä työssä tutkitaan, miten 3D web-käyttöliittymien toteuttamista voidaan helpottaa. Työn teknisenä kontribuutiona on toteutettu 3D-käyttöliittymäkomponenttikirjasto, jossa käyttöliittymän piirtäminen on toteutettu WebGL-pohjaisella 3D-moottorilla. Esimerkkisovelluksena toteutettiin 3D-ikkunointiympäristön käyttöliittymä käyttäen tässä työssä toteutettua käyttöliittymäkomponenttikirjastoa.

Työn toteutuksen yhteydessä havaittiin WebGL-pohjaisten 3D-moottorien tarjoavan palveluita hyvin vaihtelevilla abstraktiotasoilla. Lisäksi monet työssä esitellyt 3D-moottorit kehittyvät nopeasti, mikä voi aiheuttaa suuriakin muutoksia kirjaston toimintaan ja rajapintoihin. Näiden seikkojen vaikutusta 3D-käyttöliittymäkomponenttikirjaston toteutuksessa pystyttiin vähentämään tekemällä kirjastosta mahdollisimman riippumaton 3D-moottorin rajapinnoista ja palveluista.

Työn tuloksena voidaan todeta rakennetun käyttöliittymäkomponenttikirjaston helpottavan ohjelmoijan työtä. Kuitenkin toteutettu kirjasto jää ominaisuuksiltaan vielä kauas perinteisten työpöytäsovellusten toteuttamiseen tarkoitetuista vastaavissa kirjastoista.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
MATTILA, ANNA-LIISA: Composing 3D-widget library with web technologies
Master of Science Thesis, 54 pages, 4 Appendix pages
October 2012
Major: Software Engineering
Examiners: Professor Tommi Mikkonen, teaching associate Arto Salminen and researcher Jari-Pekka Voutilainen
Keywords: 3D User Interfaces, WebGL, HTML5

Building interactive web based 3D applications is possible but it is not too simple. Tools for developing interactive 3D applications are way behind from the tools used for developing interactive 2D applications. There are plenty of widget toolkits and full-fledged GUI libraries for 2D desktop application developers to use, but 3D user interfaces are still mainly made in lower abstraction level by using only tools provided by 3D engine and 3D modeling software.

In this thesis we explore how to make development of interactive web based 3D applications easier. As the technical contribution a 3D-widget library that uses a WebGL based 3D engine for rendering was designed and implemented. As an example application, a 3D-windowing environments user interface was done.

It was found that WebGL based 3D engines provide utilities in various abstraction levels and that 3D engines evolve rapidly. The latter also affects developing tools based on those engines. Effects of rapid development and changing interfaces can be minimised by making the widget library as independent from the underlying 3D-engine as possible.

As a result of the thesis work, it can be concluded that rasing the abstraction level from 3D engines primitives to 3D widgets makes it easier to develop interactive web based 3D applications. The composed library does not provide nearly as much support for user interface development as conventional desktop toolkits and GUI libraries but still it succeeds in its mission to make development of interactive web based 3D-applications more straightforward.

ALKUSANAT

Tämä diplomityö on toteutettu Tampereen teknillisessä yliopistossa Ohjelmistotekniikan laitoksella. Kiitän professori Tommi Mikkosta asiantuntevasta ohjauksesta ja hyvistä kommenteista. Kiitän myös Arto Salmista hyvistä kommenteista ja kärsivällisestä kielioppivirheiden metsästämisestä. Kiitän Jari-Pekka Voutilaista hyvistä neuvoista ja teknisestä tuesta Lively3D:n kanssa.

Kiitän myös avopuolisoani Arto Seppää oikolukuavusta, henkisestä tuesta ja kärsivällisyydestä.

Tampereella 30.8.2012
Anna-Liisa Mattila

SISÄLLYS

LYHENTEET JA TERMIT

3D	*Three dimensional*. Kolmiulotteinen.
3D-moottori	Piirtorajapinnan päälle rakennettu kirjasto, joka tarjoaa kolmiulotteisen maailman esittämiseen tarvittavia yleisiä palveluita kuten kameramatriisin operaatiot, kappaleiden muunnokset, kappaleiden väliset törmäystarkastelut sekä tuen 3D-mallien lataamiseen mallinnusohjelmista.
3D-objekti	Objekti joka koostuu 3D-mallista, eli geometrisesta muodosta, ja materiaalista, joka on sen pinnassa. Kuvaa kolmiulotteista kappaletta.
API	*Application Programming Interface*. Ohjelmointirajapinta.
CSS	*Cascading Style Sheets*. Tyylikieli, joka mahdollistaa rakenteisten dokumenttien (esim. HTML) tyylien, kuten tekstin koko ja elementtien värit, määrittämisen erillään dokumentin sisällöstä.
CSS 3D	CSS:n laajennus, joka mahdollistaa CSS-tyylejä käyttävien elementtien muunnokset 3D-avaruudessa.
Direct3D	Microsoftin 3D-grafiikan piirtämiseen tarkoitettu API. Ensisijainen 3D-piirtorajapinta Windows-ympäristöissä. Kuuluu DirectX API-kokoelmaan.
DirectX	Microsoftin ohjelmiston ja laitteiston välille kehitetty API-kokoelma. Sisältää muun muassa Direct3D-piirtorajapinnan.
DOM	*Document Object Model*. Alustariippumaton tapa esittää objektien kokoelma. Selaimessa DOM muodostaa HTML-dokumentin elementeistä puun, joka muodostaa näytettävän sivun. DOM toimii myös rajapintana selaimen ja skriptikielien välillä.
GLSL	*GL Shading Language*. Kieli, jolla OpenGL:ssa ja WebGL:ssa määritellään sävyttimet (engl. *shader*).
GNOME	*GNU Network Object Model Environmen*. GNOME on työpöytäympäristö, jota käytetään useasti esimerkiksi Linux-käyttöjärjestelmissä. GNOME:n tavoitteena on tarjota helppokäyttöinen graafinen käyttöliittymä ja kattava kehitysalusta sovelluksille.

GTK+	*GIMP toolkit*. X-ikkunointijärjestelmälle kehitetty käyttöliittymäkirjasto. Kirjasto on käytössä useissa Unix- ja Linux-järjestelmien työpöytäympäristöissä, esimerkiksi GNOME-työpöytäympäristö rakentuu GTK+:n päälle.
HTML	*HyperText Markup Language*. Merkkauskieli, jolla kuvataan tekstipohjaisen dokumentin rakenne.
JSON	*JavaScript Object Notation*. Tekstipohjainen standardi, joka tarjoaa formaatin JavaScript-pohjaiseen tiedon välitykseen. JSON määrittää mallin, jonka avulla voidaan kuvata yksinkertaisia avain–arvo-pareja.
Kirjasto	Kokoelma resursseja, esimerkiksi luokkia tai aliohjelmia, joita voidaan käyttää modulaarisen ohjelmistokehityksen apuna.
Liitännäinen	Lisäosa (engl. *plugin*) on ohjelmistokomponenttien joukko, joka tarjoaa lisäominaisuuksia suurempiin ohjelmistoihin. Esimerkiksi web-selaimeen on saatavilla liitännäisiä, jotka mahdollistavat muun muassa musiikin toiston.
Materiaali	Kuvaa 3D-objektin pintaan littyviä ominaisuuksia kuten värejä, tekstuuria, kiiltävyyttä, valon taittokerrointa jne.
MFC	*Microsoft Foundation Class Library*. C++-kirjasto, jonka läpi ohjelmoija voi käyttää Windowsin ohjelmointirajapintaa.
Motif	X-ikkunointijärjestelmälle kehitetty käyttöliittymäkomponenttikirjasto.
OpenGL	Vapaasti käytettävä laitteistoriippumaton grafiikkaohjelmointirajapinta. Ensisijainen 3D-piirtorajapinta Linux-, Unix- ja Mac-ympäristöissä. Verrattavissa Direct3D-rajapintaan.
Polygoni	Monikulmio. Tasokuvio, joka koostuu äärellisestä määrästä janoja siten, että jokaisen janan kumpikin päätepiste on jonkin toisen janan päätepiste. Janat muodostavat monikulmion sivut ja janojen päätepisteet ovat monikulmion kulmapisteitä.
SVG	*Scalable Vector Graphics*. Standardi kaksiulotteisen vektorigrafiikan ja yhdistetyn vektori- ja rasterigrafiikan kuvaamiseen XML:n avulla.

Sävytin	Engl. *Shader*. Tietokoneohjelma, joka suoritetaan näytönohjaimella. Sävyttimien avulla voidaan toteuttaa muun muassa valaistuksen, värien ja efektien laskemista. Perinteisesti sävyttimiä on kahdenlaisia: pikselisävyttimiä, jotka suoritetaan joka pikselille ja verteksisävyttimiä, jotka suoritetaan joka verteksille.
Tekstuuri	Kuva, joka näytetään 3D-kappaleen pinnassa. WebGL-tekstuurina voi toimia muun muassa kuva, video-elementti, canvas-elementti.
Verteksi	Tietorakenne, joka kuvaa pistettä 2D- tai 3D-avaruudessa. Yleensä 3D-objektit koostuvat kolmion muotoisista litteistä pinnoista, joita esitetään kulmapisteiden, verteksien, avulla.
WebGL	OpenGL ES 2.0 standardista johdettu grafiikkaohjelmointirajapinta web-sovelluksiin.
WYSIWYG	*What You See Is What You Get*. Graafinen editointiympäristö, jossa sisältö editoidessa näyttää samalta kuin lopputulos. Esimerkiksi teksinkäsittelyohjelma Microsoft Word on WYSIWYG-editori.
XML	*eXtensible Markup Language*. Standardoitu tapa esittää dokumentteja ohjelmallisesti käsiteltävässä muodossa.
Xt	*X toolkit*. X-ikkunointijärjestelmän päälle kehitetty ohjelmointirajapinta, joka tarjoaa käyttäjälle muun muassa käyttöliittymäkomponentteja.
XView	Sun Microsoftin vuonna 1988 esittelemä käyttöliittymäkomponenttikirjasto X-ikkunointijärjestelmälle.

1 JOHDANTO

Viime vuosien aikana ohjelmistokehityksen painopiste on siirtynyt perinteisistä työpöytäsovelluksista kohti web-sovelluksia [1]. Web-ohjelmointiympäristön kehitys alkoi, kun ensimmäiset web-selaimet esiteltiin 1990-luvun alkupuolella. Ensimmäisen sukupolven web-sivut olivat staattisia ja koostuivat tekstistä, kuvista ja linkeistä. Vuonna 1995 esitelty ohjelmointikieli JavaScript ja nopeasti yleistyneet liitännäiset, kuten *Flash Player*[1], sysäsivät web-sovellusten kehitystä eteenpäin. Nämä tekniikat mahdollistivat muun muassa näyttävien animaatioiden ja äänen lisäämisen web-sivuille, ja ne muuttivat web-sovellukset staattisista sivuista kohti multimediaesityksiä. Nykyiset, kolmannen sukupolven web-sovellukset, muistuttavat paljon työpöytäsovelluksia. Kolmannen sukupolven web-sovelluksia kutsutaan rikkaiksi internetsovelluksiksi (eng. *Rich Internet Application, RIA*). Rikkaissa internetsovelluksissa muun muassa vuorovaikutus on toteutettu työpöytäsovellusten käyttöliittymistä tutuilla suoran manipulaation tekniikoilla: sen sijaan, että haettaisiin uudelleen koko sivu, päivitetään vain muuttuneet elementit. [2]

Standardit, kuten HTML5 ja WebGL, parantavat web-selaimen ohjelmointiympäristöä entisestään. HTML5-standardi esittelee muun muassa *audio*- ja *video*-elementit, joiden avulla web-sivulle voidaan lisätä videoita ja ääntä ilman liitännäisiä [3]. WebGL puolestaan mahdollistaa laitteistokiihdytetyn 3D-grafiikan esittämisen selaimessa ilman liitännäisiä [4]. [1]

Vuorovaikutteisten kolmiulotteisten sovellusten toteuttaminen ei kuitenkaan ole yksinkertaista. Vuorovaikutteisten kaksiulotteisten sovellusten toteuttamista varten on olemassa pitkälle kehittyneitä käyttöliittymäkirjastoja, jotka helpottavat sovellusten kehitystyötä. Vastaavia korkean abstraktiotason käyttöliittymäkirjastoja ei kuitenkaan ole interaktiivisten 3D-sovelluksten kehitykseen. [5] [6]

Tässä työssä tutkitaan, miten 3D web-käyttöliittymien toteuttamista voidaan helpottaa. Työn teknisenä kontribuutiona toteutettiin 3D-käyttöliittymäkomponenttikirjasto, joka käyttää WebGL-pohjaista 3D-moottoria käyttöliittymän piirtämiseen. Esimerkkisovelluksena tehtiin Jari-Pekka Voutilaisen diplomityönä toteuttaman *Lively3D*[2] 3D-ikkunointiympäristön [7] käyttöliittymä uudelleen käyttöliittymäkomponenttikirjastoa hyödyntäen.

Työn rakenne on seuraava. Luvussa 2 tutustutaan yleisesti graafisiin käyttöliittymiin, niiden taustaan ja kehitystyökaluihin. Luvussa 3 käydään läpi 3D-grafiikan piirtoon liit-

[1] http://www.adobe.com/software/flash/about/
[2] http://lively3D.cs.tut.fi

tyvät perusasiat ja käsitteet sekä esitellään selaimen sisäänrakennettuja toteutustekniikoita 3D-grafiikalle. Luvussa 4 esitellään toteutettu käyttöliittymäkomponenttikirjasto. Kirjastoa käyttäen toteutettua esimerkkisovellusta käsitellään luvussa 5. Luvussa 6 kootaan yhteen työn tulokset ja johtopäätökset.

2 JOHDATUS GRAAFISIIN KÄYTTÖLIITTYMIIN

Tässä luvussa käsitellään graafisia käyttöliittymiä ja niiden taustaa, käyttöliittymien ohjelmoimiseen liittyviä abstraktiotasoja ja sitä, millaisia erikoispiirteitä käyttöliittymien ohjelmointiin liittyy. Lisäksi esitellään esimerkinomaisesti käyttöliittymäkirjasto *wxWidgets*.

2.1 Taustaa

Käyttöliittymä on ihmisen ja sovelluksen välisen vuorovaikutuksen mahdollistava rajapinta. Sen tehtävänä on muuntaa käyttäjän syötteet ja toiminnot tietokoneohjelman ymmärtämään muotoon, ja tulkata tietokoneohjelmalta saadut viestit käyttäjän ymmärtämään muotoon. Graafinen käyttöliittymä on käyttöliittymä, joka koostuu graafisista elementeistä, ja sitä käytetään yleensä hiirtä (tai vastaavaa osoitinlaitetta) ja näppäimistöä käyttäen. Graafisten käyttöliittymien kehitys alkoi 1980-luvulla sen jälkeen, kun tietokonehiiri keksittiin. Graafisten käyttöliittymien myötä alettiin kiinnittää myös enemmän huomiota käyttöliittymäsuunnitteluun ja käytettävyystutkimus alkoi. [5]

Työpöytäsovellusten käyttöliittymissä on yleistä, että samalla alustalla ajettavien sovellusten käyttöliittymät näyttävät samalta. Windows-sovelluksilla on oma ulkonäkönsä ja tuntumansa, ja Mac OS -sovelluksilla omansa. Työpöytäsovellusten kehittämiseen tarkoitetut käyttöliittymätyökalut ovat pitkälle kehittyneitä, ja modernit työpöytäympäristöt tarjoavat ennalta määritellyn näköiset käyttöliittymäkomponentit sovellusohjelmoijalle, jotta yhtenäisen näköisten käyttöliittymien toteuttaminen olisi mahdollisimman yksinkertaista.

Graafiset käyttöliittymät eivät rajoitu pelkästään työpöytäsovellusten, mobiilisovellusten ja web-sovellusten perinteisiin kaksiulotteisiin käyttöliittymiin. Kirjassa *3D User Interfaces: Theory and Practice* [5] määritellään 3D-käyttöliittymä seuraavasti: 3D-käyttöliittymä on käyttöliittymä, joka sisältää kolmiulotteista vuorovaikutusta. Kolmiulotteinen vuorovaikutus tarkoittaa sitä, että käyttäjä kommunikoi sovelluksen kanssa suoraan 3D-ympäristön kautta – esimerkiksi käyttäjä valitsee hiirellä 3D-maailman objektin navigoidakseen sen luokse. Vastaava tilanne voitaisiin toteuttaa kaksiulotteisella käyttöliittymällä siten, että käyttäjä valitsisi kaksiulotteisesta objektilistauksesta sen kolmiulotteisen maailman objektin, jonka luokse hän haluaa navigoida.

3D-käyttöliittymien sovellusalueena ovat perinteisesti olleet erilaiset virtuaaliympäristöt. Virtuaaliympäristö on synteettinen maailma, jota katsotaan yleensä ensimmäisen

persoonan näkökulmasta. Käyttäjä pystyy vuorovaikutukseen ympäristön kanssa reaaliaikaisesti. Virtuaaliympäristöinä voidaan pitää myös joitain 3D-tietokonepelien maailmoja, vaikka niissä käyttäjän kontrolli ympäristöön on monesti rajoitettua. [5]

Virtuaaliympäristö voidaan esittää 3D-grafiikkana tavallisella tietokoneen näytöllä, mutta sen esittämiseen voidaan käyttää myös erilaisia apulaitteita kuten 3D-näyttöjä, 3D-laseja ja erikoistuneita osoitinlaitteita ja liiketunnistinsensoreita [5]. Tässä työssä käsitellään virtuaaliympäristöjä, joiden esitystapa on perinteinen 2D-näytölle projisoitu maailma, jota käyttäjä käyttää perinteisillä kontrollilaitteilla, kuten hiirellä ja näppäimistöllä.

Graafiset käyttöliittymät ovat tietokoneen käyttäjälle nykyisin arkipäivää. Kuitenkin graafisten käyttöliittymien ja sovellusten alle kätkeytyy paljon ohjelmallisesti monimutkaisia asioita, joita käsitellään tarkemmin kohdissa 2.2 ja 2.3.

2.2 Abstraktiotasot

Graafisen käyttöliittymän ohjelmoiminen nykyaikaisia käyttöliittymäkehitystyökaluja käyttäen on melko suoraviivaista. Käyttöliittymän ulkoasun voi suunnitella graafisella editorilla, jossa käyttäjä näkee jo muokkausvaiheessa, miltä lopputulos tulee näyttämään (WYSIWYG-editori) tai määritellä rakenteisten dokumenttien, kuten XML, avulla. Käyttöliittymän toiminnot voidaan sitoa käyttöliittymäkomponentteihin korkean tason rajapintojen avulla. Kun ensimmäisiä graafisia käyttöliittymiä ohjelmoitiin, ei moderneja työkaluja kuitenkaan ollut. Tällöin graafisen käyttöliittymän toteuttamiseen liittyi oleellisena osana muun muassa matalan tason grafiikkaohjelmointi.

Grafiikkaohjelmointia matalalla abstraktiotasolla on esimerkiksi suorapiirto, jossa tietokoneen näytöllä näytettävää kuvaa käsitellään kaksiulotteisena pikselitaulukkona. Grafiikkaohjelmoinnin näkökulmasta suorapiirto on sama kohdassa 3.1 esiteltävän 3D-piirtoliukuhihnan rasterointivaiheen kanssa.

Käyttöliittymän toteuttamiseen tai yleensäkään interaktiivisen dynaamisen grafiikan toteuttamiseen suorapiirto ei tarjoa mitään valmiina. Ohjelmoijan täytyy pitää kirjaa siitä, mitä näytöllä milloinkin näytetään – esimerkiksi, jos kaksi objektia on päällekkäin, kumpi piirretään. Lisäksi ohjelmoijan täytyy huolehtia vuorovaikutuksesta käyttäjän kanssa, eli laskea mihin pikseliin esimerkiksi hiiren klikkaus osui, ja onko tämä pikseli osa aluetta, joka ottaa klikkauksia vastaan. [8]

Kaikki tietokoneen näytöllä esitettävä grafiikka on viime kädessä piirretty pikseli kerrallaan. Grafiikkaohjelmoinnin helpottamiseksi on kehitetty piirtorajapintoja grafiikkalaitteiston ja sovelluksen välille. Piirtorajapinnat nostavat abstraktiotasoa pikseleistä 2D- ja 3D-objekteihin. Piirtorajapintojen 2D- ja 3D-objektit ovat yleensä geometrisia muotoja, jotka koostuvat polygoniverkosta. Muun muassa *OpenGL*[1] on piirtorajapinta.

Graafisten käyttöliittymien yleistymisen mahdollisti ikkunointijärjestelmät ja työpöy-

[1] OpenGL, http://www.opengl.org/

täympäristöt. Ikkunointijärjestelmä toteuttaa sen osan sovelluslogiikkaa, joka liittyy sovellusikkunan esitystavan määrittämiseen ja näyttämiseen ja se huolehtii myös käyttäjän vuorovaikutuksesta ikkunan kanssa. Ikkunointijärjestelmän olemassaolo helpottaa useiden yhtä aikaa esillä olevien sovellusikkunoiden hallitsemista, mutta yksittäisen sovelluksen käyttöliittymän ohjelmoinnissa säilyvät omat haasteensa. Ohjelmoija joutuu edelleenkin huolehtimaan käyttöliittymäkomponenttien piirrosta suorapiirron tai piirtorajapinnan avulla ja toteuttamaan käyttöliittymäkomponenttien vuorovaikutuksen käyttäjän kanssa. [8] [9]

Ikkunointijärjestelmien päälle on kehitetty työpöytäympäristöjä, jotka tarjoavat myös työkaluja käyttöliittymien ohjelmointiin. Työpöytäympäristökohtaiset käyttöliittymäkomponenttikirjastot sisältävät hierarkkisen käyttöliittymäkomponenttivalikoiman, jossa komponenttien esitystapa on ennalta määritelty. Ohjelmoijan tehtäväksi jää rakentaa komponenteista haluamansa kokonaisuus ja toteuttaa komponenteille tapahtumakäsittelijät, eli toimenpiteet, jotka suoritetaan, jos esimerkiksi nappia painetaan. Kommunikointi piirtorajapinnan tai piirtolaitteiston kanssa ei ole enää graafisen käyttöliittymän ohjelmoijan tehtävä. [9]

Käyttöliittymäkomponentit koostuvat yleensä resursseista, joita ovat graafinen esitystapa, tyyppi ja tapahtumakäsittelijät. Graafinen esitystapa tarkoittaa kappaleen väriä, muotoa, fonttia ja muita kappaleen komponentin esitystapaan liittyviä ulkoisia ominaisuuksia. Komponentin tyyppi puolestaan määrittää komponentin paikan komponenttihierarkiassa, ja usein se määrittelee myös osan graafisesta esitystavasta, esimerkiksi muodon. Tapahtumakäsittelijät ovat yleensä ohjelmoijan määrittelemiä funktioita, joiden avulla komponentti reagoi käyttäjän syötteisiin. Komponentin resurssit on tavallisesti peritty yleiseltä kaikille komponenteille yhteiseltä kantaluokalta, mutta eri tyyppiset komponentit voivat lisäksi määrittää uusia resursseja tai ylikirjoittaa perittyjä resursseja. [9]

Komponenttikirjastot ja niihin liittyvät muut käyttöliittymätyökalut, esimerkiksi graafiset editorit, alkoivat kehittyä 80-luvulla graafisten käyttöliittymien yleistymisen myötä. Nykyiset työpöytäympäristöt, kuten *Windows* ja *GNOME*, tarjoavat ohjelmistokehittäjälle käyttöliittymien ohjelmointiin perustyökalut, kuten joukon käyttöliittymäkomponentteja. Työpöytäympäristöjen sisäänrakennettujen käyttöliittymätyökalujen päälle on lisäksi kehitetty käyttöliittymäkirjastoja, jotka pyrkivät mahdollistamaan alustariippumattomien käyttöliittymien ohjelmoinnin. Nämä kirjastot käyttävät työpöytäympäristöjen tarjoamia käyttöliittymäkomponentteja, silloin kun se on mahdollista, saavuttaakseen työpöytäympäristön tutun käyttöliittymän ulkonäön ja tuntuman. Tällainen kirjasto on esimerkiksi kohdassa 2.4 esiteltävä *wxWidgets*.

Työpöytäsovellusten käyttöliittymätyökalujen kehitys on kulkenut käsikädessä ikkunointijärjestelmien ja työpöytäympäristöjen kehityksen kanssa. Nykyiset työkalut ovat pitkälle kehittyneitä ja abstrahoivat muun muassa grafiikkaohjelmoinnin ja tapahtumakäsittelyn mahdollisimman pitkälle. 3D-käyttöliittymien vastaavia kehitystyökaluja ei kui-

tenkaan juuri ole. Syy tähän lienee se, että perinteiset työpöytäympäristöt ja ikkunointijärjestelmät ovat kaksiulotteisia.

3D-käyttöliittymät tehdään nykyään pitkälti samoilla työkaluilla kuin 3D-pelit, eli käyttäen 3D-moottoria tai piirtorajapintaa apuna piirrossa ja mallintamistyökaluja, esimerkiksi *Blenderiä*[2], 3D-mallien luomisessa [5]. Tämä mahdollistaa elementtien käsittelyn 3D-moottorin abstraktiotasolla, mutta ei käyttöliittymäkomponentteina, kuten nykyiset 2D-käyttöliittymätyökalut mahdollistavat. Kuitenkin käyttöliittymäkomponenttikirjastojen perusajatus, hierarkkiset komponentit joiden esitystapa on ennalta määrätty, on siirrettävissä myös kolmiulotteiseen maailmaan. Vaikka esimerkiksi kolmiulotteinen ikkuna näyttäisi erilaiselta kuin perinteinen kaksiulotteisen työpöytäsovelluksen ikkuna, komponentin käyttötarkoitus on sama. [6]

2.3 Ohjelmointimalli

Tässä kohdassa esitellään interaktiivisen graafisen sovelluksen toteutukseen liittyviä erikoispiirteitä ja niihin yleisesti sovellettuja ratkaisuja ja suunnittelumalleja. Suunnittelumalleja on olemassa lukuisia, mutta käymme läpi niistä vain oleellisimmat graafisia interaktiivisia sovelluksia ajatellen.

2.3.1 Tapahtumapohjainen ohjelmointi

Interaktiivinen sovellus saa käyttäjältä syötteitä suorituksen aikana. Sitä, milloin tai missä järjestyksessä syötteet tulevat, ei voida tietää etukäteen. Interaktiivisen sovelluksen ohjelmoinnissa täytyykin ottaa huomioon, että ohjelman suoritusjärjestys ei ole ennalta määrätty eikä käyttäjän syötteiden saapumisajankohtaa voida tietää.

Tapahtumapohjaisen ohjelmoinnin perusajatus on se, että sovellukselle välitetään *tapahtumia*, joita voivat synnyttää esimerkiksi käyttäjän toimet, kuten hiiren klikkaus tai näppäimistön napin painallus, erilaiset ajastimet, käyttöjärjestelmä ja jopa sovellus itse. Tapahtuman ottaa vastaan sovelluksessa *tapahtumakäsittelijä*. Tapahtumakäsittelijät (tapahtumakuuntelijat) ovat funktioita, jotka on rekisteröity ottamaan vastaan tiettyjä tapahtumia. [10]

Tapahtumiin liittyy tapahtuman synnyttäjän ja tapahtumankäsittelijän lisäksi tapahtumaolio. Tapahtumaolio välitetään tapahtuman synnyttäjältä tapahtumankäsittelijälle ja se sisältää tietoa liittyen tapahtumaan. Esimerkiksi näppäimistötapahtuman tapahtumaolio sisältää yleensä vähintään tiedon siitä, mitä näppäintä painettiin. [10]

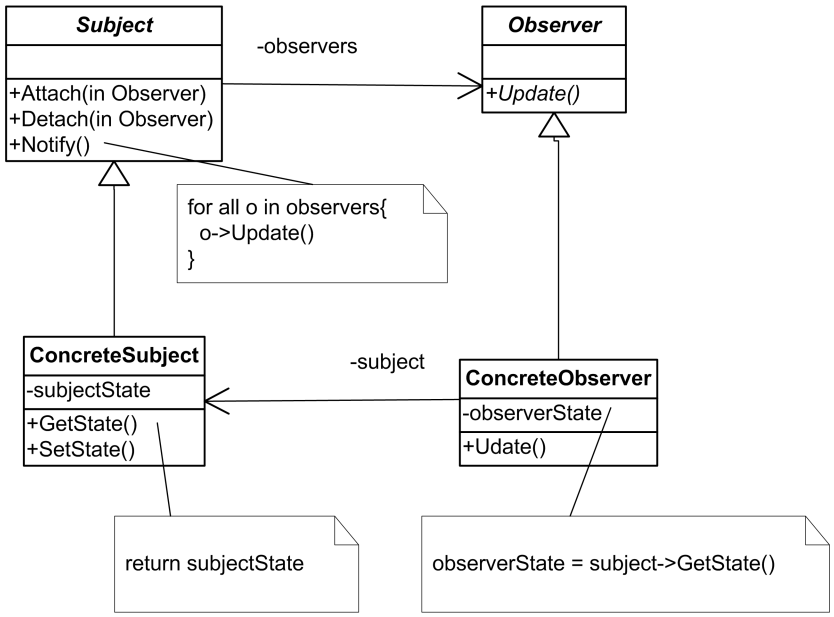
Tapahtumapohjainen ohjelmointiparadigma mahdollistaa käyttäjän syötteisiin reagoimisen yksinkertaisesti. Sovelluksen ei tarvitse aktiivisesti kysellä (eng. *polling*), onko nappia painettu, vaan napin painalluksesta muodostuu tapahtuma, jolla tieto napin painamisesta välittyy rekisteröidyille tapahtumakäsittelijöille. [10]

[2] http://www.blender.org/

2.3.2 Tarkkailija-suunnittelumalli

Tarkkailija-suunnittelumalli määrittelee olioiden välille yhden suhde moneen riippuvuuden. Kun yhden olion tila muuttuu, osaavat siitä riippuvat oliot päivittää itsensä automaattisesti. [11]

Kuvassa 2.1 on esitetty Tarkkailija-suunnittelumalli luokkakaaviona. Luokkakaaviossa *Subject* on olio, josta *Observer*-oliot, eli tarkkailijat, ovat riippuvaisia. Subject-olio lähettää *notify()*-metodilla tiedon kaikille sen tarkkailijoille muutoksesta, jonka jälkeen tarkkailijat kysyvät Subject-oliolta tilapäivitystä. Kuvassa Subject- ja Observer-oliot ovat abstrakteja kantaluokkia jotka määrittävät yhtenäiset rajapinnat tarkkailijoille ja tarkkailtaville olioille.



Kuva 2.1: Tarkkailija-suunnittelumalli esitettynä luokkakaaviona [11]

Tarkkailija-suunnittelumalli voidaan toteuttaa edellä esiteltyjen tapahtumien avulla. Subject-olio voidaan mieltää tapahtuman synnyttäjäksi ja Observer-olio tapahtuman käsittelijäksi. [10] [11]

2.3.3 MVC-suunnittelumalli

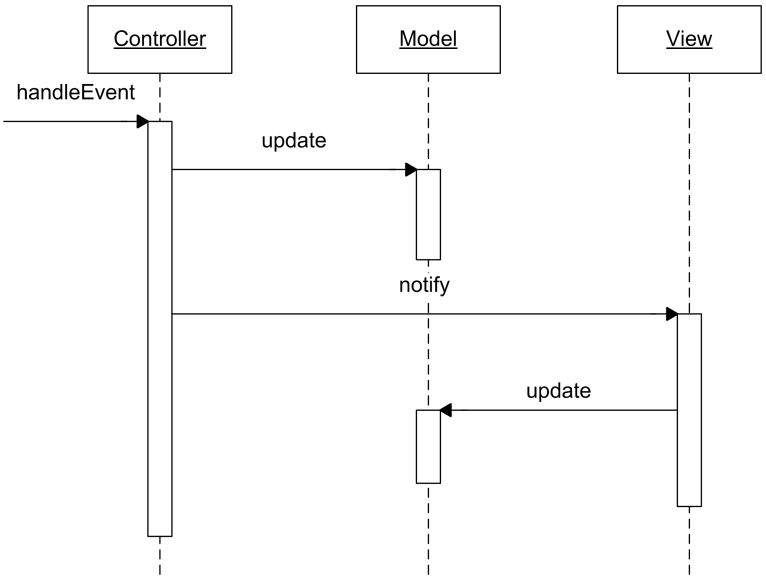
Sovelluslogiikan ja käyttöliittymäkoodin eriyttäminen toisistaan on tärkeää sovelluksen ylläpidettävyyden kannalta, mutta myös käyttöliittymän räätälöinnin kannalta –samaan sovellukseen halutaan usein tehdä monta erilaista käyttöliittymää, esimerkiksi mobiilikäyttöliittymä ja työpöytäkäyttöliittymä.

Sovelluslogiikan ja käyttöliittymän eriyttäminen toteutetaan usein käyttäen *MVC*-suunnittelumallia (*Model View Controller*). MVC-mallissa sovellus on jaettu kolmeen

erilliseen osaan: sovelluksen yleiseen logiikkaan (*Model*), joka sisältää muun muassa ohjelman tilatiedon ja datan, näkymään, joka näytetään käyttäjälle (*View*) ja käyttäjän syötteisiin reagointiin eli kontrolleihin (*Controller*). [11]

MVC-malli mahdollistaa käyttöliittymän ulkoasun räätälöinnin lisäksi myös kontrollien räätälöinnin. Kontrolli määrittää sen, miten ohjelma reagoi käyttäjän syötteisiin, eli mitä toimintoja suoritetaan, kun esimerkiksi nappia painetaan. MVC-malli erottaa kontrollilogiikan yleisestä sovelluslogiikasta ja helpottaa näin ollen ylläpidettävyyttä. [11]

Kuvassa 2.2 on esitetty MVC-mallin toimintaperiaate sekvenssikaaviona. Kontrolliosa ottaa vastaan käyttäjän syötteitä ja muokkaa syötteiden perusteella ohjelman mallia. Tämän jälkeen kontrolliosa ilmoittaa näkymälle, että malli on päivitetty. Lopuksi näkymä päivittää itsensä mallin mukaiseksi.

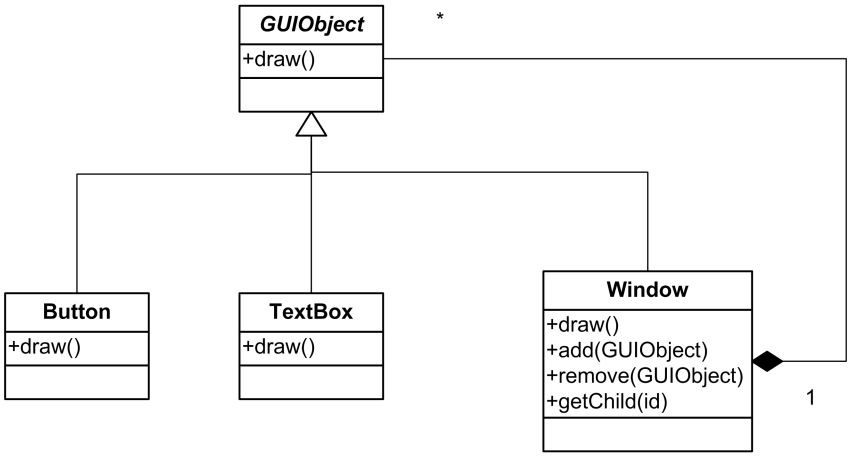


Kuva 2.2: MVC-suunnittelumalli

2.3.4 Rekursiokooste

Rekursiokoostetta käytetään hierarkkisen, puumaisen, rakenteen aikaansaamiseksi. Graafisten käyttöliittymien tapauksessa rekursiokoostetta voidaan hyödyntää ikkunoiden toteuttamisessa. [11]

Kuvassa 2.3 on esitetty rekursiokooste luokkakaaviona. Kuvan esimerkissä kaikki komponentit, *Button*, *TextBox* ja *Window* periytyvät yhteisestä abstraktista kantaluokasta *GUIObject*. *Window*-objekti koostuu lisäksi myös muista *GUIObject*-tyyppisistä objekteista. Rekursiokoosteella saadaan aikaiseksi se, että ikkuna voi sisältää muita käyttöliittymäkomponentteja, kuten nappeja, tekstikenttiä ja ikkunoita lapsinaan ja rajapinta lapsikomponenttien käsittelyyn on yhtenäinen.



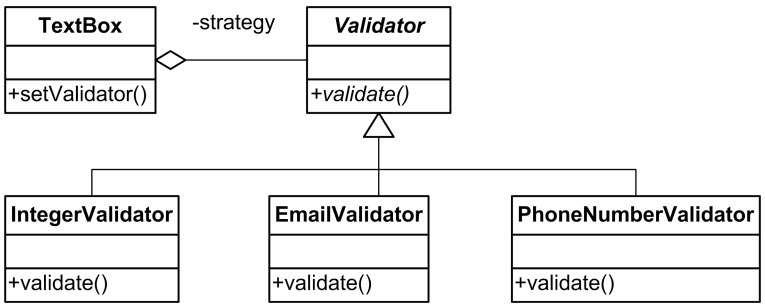
Kuva 2.3: Esimerkki rekursiokoosteesta luokkakaaviona

2.3.5 Strategia-suunnittelumalli

Strategia-suunnittelumalli määrittelee algoritmiperheen ja abstrahoi siihen kuuluvat algoritmit siten, että ne ovat keskenään vaihdettavia. Algoritmiperheen algoritmeja voidaan muuttaa niin, että sovellusta joka käyttää algoritmia ei tarvitse muuttaa. [11]

Kuvassa 2.4 on esitetty strategia-suunnittelumalli luokkakaaviona käyttäen esimerkkinä syötteen validointia. Kuvassa erilaiset validaattoriluokat on periytetty abstraktista kantaluokasta *Validator*, jonka ansiosta validaattoreita voidaan käyttää yhtenäisen rajapinnan kautta. Validaattoria käyttävä luokka koostuu validaattoreista ja se voi asettaa käyttämänsä validaattorin metodilla *setValidator*.

Käyttäjän syötteen validointi on oleellinen osa interaktiivisten ohjelmien toteutusta. Validoinnin avulla voidaan varmistua siitä, että oikeat tiedot syötetään oikeisiin kohtiin, esimerkiksi sähköpostikenttään ei pysty syöttämään vahingossa puhelinnumeroa. Syötteen validoimattomuudella voi olla myös vakavia seurauksia: esimerkiksi jos käyttäjä kirjoittaa web-sovelluksen tekstikenttään JavaScript-koodia ja sovellus tallettaa datan sellaisenaan, on mahdollista, että käyttäjän ohjelmakoodi päätyy suoritettavaksi.



Kuva 2.4: Strategia-suunnittelumallin luokkakaavioesimerkki

2.4 Esimerkkikirjasto: wxWidgets

WxWidgets on C++ -kirjasto, joka mahdollistaa sellaisten käyttöliittymien tekemisen, jotka toimivat Windows-, MacOS-, Linux- ja Unix-alustoilla. Kirjasto on valittu tähän työhön esimerkiksi modernista korkean tason käyttöliittymäkirjastosta, joka on laajalti käytetty ja yhteensopiva monien ikkunointijärjestelmien kanssa. Kirjaston esittelyn tarkoitus on tuoda käsitys siitä, minkälaisia nykyiset työpöytäkäyttöliittymien toteuttamiseen tarkoitetut käyttöliittymäkirjastot ovat.

2.4.1 Taustaa

WxWidgets sai alkunsa vuonna 1992 Edinburghin yliopistossa projektissa, jossa tarvittiin käyttöliittymäkirjasto, joka toimii sekä Windows- että Unix-ympäristöissä. Alunperin wxWidgets rakennettiin siten, että se toimi yhteen sekä *XView*-kirjaston, joka on X-ikkunointijärjestelmän käyttöliittymäkomponenttikirjasto, että Microsoftin *MFC*-kirjaston, joka mahdollistaa Windowsin tarjoaman ohjelmointirajapinnan käyttämisen C++:lla, kanssa. Myöhemmin MFC:n käyttö vaihdettiin puhtaaksi *Win32*-API:n käytöksi ja Xview-kirjaston lisäksi Linux- ja Unix-alustoja varten kirjoitettiin versio, joka käytti *Motif*-kirjastoa, joka on *XView*:n tapaan X-ikkunointijärjestelmälle kehitetty käyttöliittymäkomponenttikirjasto. Ajan myötä wxWidgets sai innokkaan käyttäjäkunnan, joka myös kehitti kirjastoa eteenpäin. Vuonna 1995 Markus Holzem julkaisi kirjastosta *Xt*-version. *Xt* on X-ikkunointijärjestelmän päälle kehitetty työkalukirjasto, jonka päälle *Motif* on rakennettu. *Xt*-versio wxWidgetsistä mahdollisti kirjaston käytön X-ikkunointiympäristössä ilman Motif-kirjastoa, joka on kaupallinen. Kirjastosta kehitettiin myös Mac-versio. Vuonna 1997 wxWidgets kirjastosta julkaistiin toinen versio, jossa rajapintoja ja C++:n käyttöä pyrittiin parantamaan. Myös *GNOME*-työpöytäympäristö teki tuloaan vuonna 1997. GNOME käytti *X11*-ikkunointiympäristön päälle rakennettuja GTK+-komponentteja. Vuonna 1998 wxWidgets 2:sta tehtiin *GTK+*-komponentteja käyttävä versio. Samalla päätettiin, että wxWidgets 2:sta ei tehdä *Xt*-versiota, sen sijaan *Motif*-versio tehtiin. Tällä hetkellä wxWidgets tukee ympäristöjä, jotka on esitelty taulukossa 2.1. [12]

Jos mahdollista, wxWidgets käyttää työpöytäympäristön tarjoamia komponentteja, jotta käyttöliittymän tyyli ja tuntuma olisi mahdollisimman samanlainen kuin muiden samaa alustaa käyttävien sovellusten. wxWidgets sisältää myös oman toteutuksensa käyttöliittymäkomponenteista joita voidaan käyttää, jos työpöytäympäristön tarjoamia komponentteja ei ole käytettävissä tai niitä ei haluta käyttää.

Vaikka wxWidgets on C++-kirjasto, sitä voidaan käyttää myös muilla ohjelmointikielillä. Tällä hetkellä wxWidgetsiin on tehty rajapinnat muun muassa Pythonille, Perlille ja C#:lle. [12]

Taulukko 2.1: wxWidgetsin tukemat ympäristöt [12]

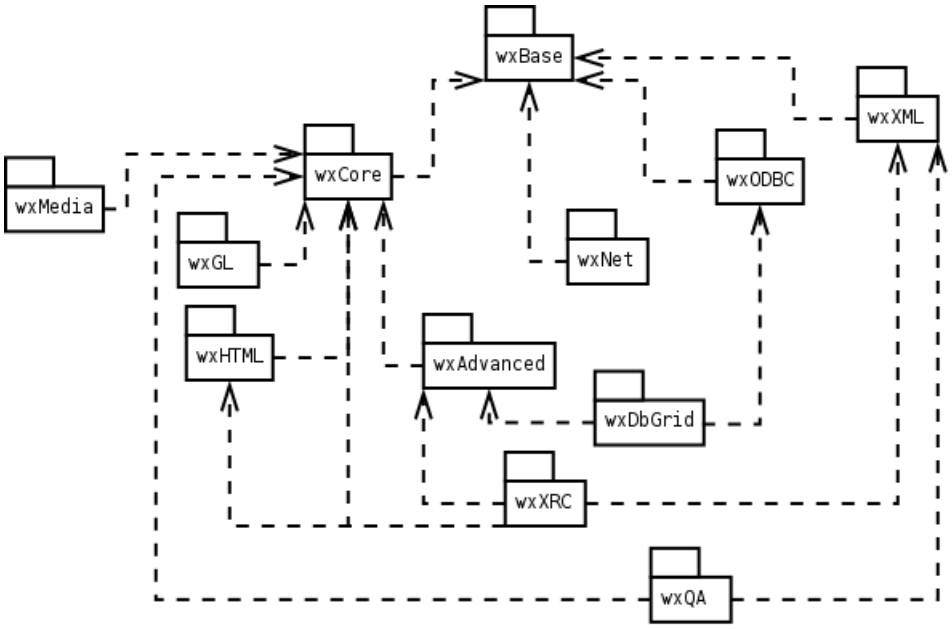
Nimi	Ympäristö
wxGTK	GTK:ta käyttäviin Linux ja Unix järjestelmiin. Käyttää GTK+:n käyttöliittymäkomponentteja.
wxMSW	32- ja 64-bittisille Windows ympäristöille.
wxWinCE	Windows CE/Windows Mobile-pohjaisille laitteille. Sisältyy wxMSW versioon.
wxMac	Mac OS 9 ja Mac OS X 10.2 järjestelmistä eteenpäin. Käyttää Carbon-API:a.
wxX11	X11 ikkunointiin pohjautuville Unix ja Linux järjestelmille. Käyttää wxWidgetsin omia komponentteja.
wxMGL	SciTech Software Inc. yrityksen MGL työkaluille.
wxMotif	Linux ja Unix järjestelmille, jotka käyttävät OpenMotifia tai Lesstifia.
wxCocoa	Mac OS X järjestelmiin. Käyttää Cocoa-API:a.
wxOS2	OS/2 järjestelmille.
wxPalmOS	PalmOS laitteille.

2.4.2 Kirjaston rakenne

WxWidgets koostuu useista kirjastoista, jotka riippuvat toisistaan kuvan 2.5 mukaisesti. Kaikki wxWidgetsiä käyttävät ohjelmat tarvitsevat *wxBase*-kirjastoa. Graafista käyttöliittymää tehtäessä myös *wxCore*-kirjasto on pakollinen. Muut osat ovat apukirjastoja, joiden käyttö on vapaaehtoista. wxWidgets voidaan kääntää yhtenä kokonaisuutena kaikkine osineen, mutta ohjelmoija voi myös valita, mitä kirjastoja hän tarvitsee ja kääntää vain ne. [13]

Kirjasto *wxBase* tarjoaa tuen alustariippumattomien komentorivisovellusten toteuttamiselle, ja *wxCore* sisältää graafisen käyttöliittymän ohjelmointiin tarvittavat komponentit. Graafisen käyttöliittymän ohjelmointiin on lisäksi kirjastot *wxGL*, *wxMedia*, *wxHTML* ja *wxAdvanced*. Kirjastot *wxNet*, *wxODBC* ja *wxXML* ovat yleisiä apuvälineitä käyttöliittymien, myös komentorivipohjaisten, ohjelmointia varten. Kirjasto *wxGL* mahdollistaa OpenGL:n käyttämisen käyttöliittymäkomponenttien piirrossa. *WxGL* ei ole mukana kokonaiskäännöksessä, vaan se tarvitsee aina kääntää erillisenä osana, jos sitä halutaan käyttää. *WxMedia* sisältää erilaisia multimediatyökaluja ja *wxHTML* sisältää muun muassa HTML-piirtoluokan, jonka avulla voidaan tulkata ja piirtää HTML-kuvauksen mukaisia näkymiä. *WxAdvanced* sisältää edistyneitä ja harvoin käytettyjä graafisen käyttöliittymän toteuttamiseen liittyviä luokkia. Kirjasto *wxNet* sisältää työkaluja verkkoyhteyden käyttämiseen. Kirjasto *wxODBC* on tietokantakirjasto ja kirjasto *wxXML* sisältää XML:n tulkkaamiseen tarvittavat työkalut. Lisäksi wxWidgets sisältää kirjastot *wxDbGrid*, *wxXRC* ja *wxQA*. Kirjasto *wxDbGrid* yhdistää tietokantakirjaston ja *wxAdvanced* kirjaston ominaisuuksia, *wxXRC* tarjoaa työkalut käyttöliittymäkomponenttien määrittämiseen XML:n

avulla ja *wxQA* tarjoaa työkaluja laadunvarmistukseen. [13]



Kuva 2.5: wxWidgets kirjastojen riippuvuudet [13]

2.4.3 Käyttöliittymäkomponentit

WxWidgets tarjoaa valmiita käyttöliittymäkomponentteja ikkunoiden, valikkojen, valitsimien ja kontrollien, joita ovat muun muassa napit, liu'ut ja valintaruudut (engl. *checkbox*), toteuttamiseen. Kaikki wxWidgetsin luokat ovat perittyjä *wxObject*-luokasta. Luokka toteuttaa yleiset operaatiot, kuten esimerkiksi instanssien samuuden vertailun. Ikkunoille wxWidgets tarjoaa *wxObject*- ja *wxEventHandler*-luokista perityn kantaluokan *wxWindow*. Lisäksi esimerkiksi kontrolleille on oma kantaluokka *wxControl*.

WxWidgets-sovellus rakentuu hierarkkisesti siten, että jokaisella sovelluksella on oltava pääikkuna, joka luodaan, kun sovellus alustetaan. Lapsikomponentit liitetään pääikkunaan, tai johonkin pääikkunan lapsi-ikkunaan, komponentin luomisvaiheessa. Sovelluksen komponentit muodostavat siis puumaisen hierarkian, jossa pääikkuna on juurena.

WxWidgets sisältää valmiiden käyttöliittymäkomponenttien lisäksi myös niin sanottuja näkymättömiä komponentteja, joiden tarkoituksena on helpottaa käyttöliittymien ohjelmointia. Tällaisia apuluokkia ovat muun muassa validaattori- ja tietorakenneluokat sekä pohjapiirustusluokka, joka hallitsee komponenttien sijoittelua ikkunan sisällä. Pohjapiirustusluokka kertoo siihen liitetylle ikkunalle, miten pohjapiirustusluokkaan määritellyt komponentit sijoitellaan, minkä kokoisina komponentit näytetään ja kuinka paljon komponenttien välissä on tyhjää tilaa. Pohjapiirustusluokka huolehtii komponenttien uudelleensijoittelusta kun esimerkiksi ikkunan kokoa muutetaan. [13]

Omien käyttöliittymäkomponenttien luominen onnistuu laajentamalla valmiita

wxWidgets-komponentteja periytymisen avulla. WxWidgets mahdollistaa vapaan periytymishierarkian, eli kaikki wxWidgetsin tarjoamat komponentit ovat laajennettavissa. [13]

Omien komponenttien periyttämisessä wxWidgetsin komponenteista täytyy huomioida yksi asia: wxWidgets ei käytä virtuaalifunktioita polymorfismin aikaansaamiseksi vaan se toteuttaa polymorfisin omilla RTTI-makroilla. RTTI on lyhenne englanninkielisestä termistä *run-time type identification*, joka tarkoittaa polymorfismin mahdollistavaa ajonaikaista tyypin määritystä. [13]

2.4.4 Vuorovaikutus käyttäjän kanssa

WxWidgetsin käyttöliittymäkomponentit, jotka reagoivat käyttäjän syötteisiin, periytyvät luokasta *wxEvent* ja sisältävät tapahtumataulun, joka määrittää mitä tapahtumia komponentti ottaa vastaan ja mitä funktiota tapahtuman tullessa komponentille kutsutaan. WxWidgets saa tapahtumat, esimerkiksi hiiren klikkaukset, ikkunointijärjestelmältä. Tapahtumalle lähdetään etsimään käsittelijää ensin tapahtuman saaneen ikkunan lapsikomponenteista ja edelleen tämän lapsista, kunnes sopiva käsittelijä löytyy tai kaikki tapahtumataulut on käyty läpi. [13]

Tapahtumakäsittelijöiden rekisteröinti, eli sitominen tapahtumiin, voidaan tehdä kahdella tapaa, joko makrojen avulla tai määrittämällä tapahtumakäsittelijät ohjelman suorituksen aikana funktiokutsun avulla. Tapahtumakäsittelijäksi voidaan sitoa joko wxWidgetsin valmis funktio tai itse määritelty funktio. WxWidgets mahdollistaa myös omien tapahtumien luomisen tapahtumaluokista periyttämällä. [13]

3 SELAIMET JA 3D-GRAFIIKKA

Tässä luvussa käydään läpi perusteet 3D-grafiikan piirrosta ja tutustutaan tekniikoihin joilla 3D-grafiikkaa voidaan toteuttaa web-selaimessa. Luvussa esitellään myös 3D-moottoreita, joiden tarkoitus on helpottaa 3D-sovellusten kehittämistä.

3.1 3D-grafiikan perusteet

Tässä kohdassa kuvaillaan 3D-grafiikan piirron perusvaiheet, jotka muodostavat perinteisen 3D-piirtoliukuhihnan. Luku on kirjoitettu Antti Puhakan teoksen *3D-grafiikka* [14] pohjalta.

3D-piirrossa muodostetaan kolmiulotteisesta mallista mahdollisimman realistisen näköinen yleensä 2D-tasoon projisoitu kuva. 3D-grafiikan piirtovaiheita kuvataan usein liukuhihnana, jonka katsotaan koostuvan kolmesta päävaiheesta: sovellusvaihe, geometriavaihe ja rasterointivaihe. Piirtorajapinnat, esimerkiksi OpenGL, abstrahoivat usein geometriavaiheen ja rasterointivaiheen rajapintojen taakse, jolloin ohjelmoijan toteutettavaksi jää vain sovellusvaihe.

3D-grafiikka pohjautuu pitkälti koordinaatistomuunnoksiin, joita on tapana esittää 4x4-matriisien kertolaskuina. 3D-grafiikkaan liittyy myös monenlaisia koordinaatistoja. Maailmakoordinaatisto on koordinaatisto, jossa origo on maailman keskipisteessä. Mallin koordinaatistossa origo on mallin keskipisteessä. Katsojan koordinaatistolla puolestaan tarkoitetaan koordinaatistoa jossa kamera eli katsoja, on origossa. Ikkunakoordinaatisto (kuvaruutukoordinaatisto) on koordinaatisto, jossa näytön keskipiste on origossa.

Eri vaiheet piirtoliukuhihnasta käsittelevät maailmaa eri koordinaatistoissa. Sovellusvaiheessa käsitellään yksittäisiä kappaleita mallin koordinaatistossa. Sovellusvaiheessa luodaan aluksi kolmiulotteinen malli, joka muodostuu pisteistä, viivoista ja monikulmioista. Kun malli on muodostettu, sille määritetään translaatio, rotaatio, skaalaus, valonlähteet ja suoritetaan mahdollinen animointi sekä fysiikan laskeminen. Malliin liittyy yleensä myös oheisdataa, kuten materiaali. Materiaali määrittelee mallin pinnan ominaisuuksia, joita ovat muun muassa pinnassa mahdollisesti näytettävä kuva eli tekstuuri, pinnan kiiltävyys, väri ja pinnan valontaittokerroin. Sovellusvaihe suoritetaan yleensä tietokoneen suorittimella ja sitä ei aina edes mielletä osaksi piirtoliukuhihnaa. Sovellusvaiheessa muodostetun mallin tiedot lähetetään keskusyksiköltä näytönohjaimelle, jossa muut liukuhihnan vaiheet suoritetaan.

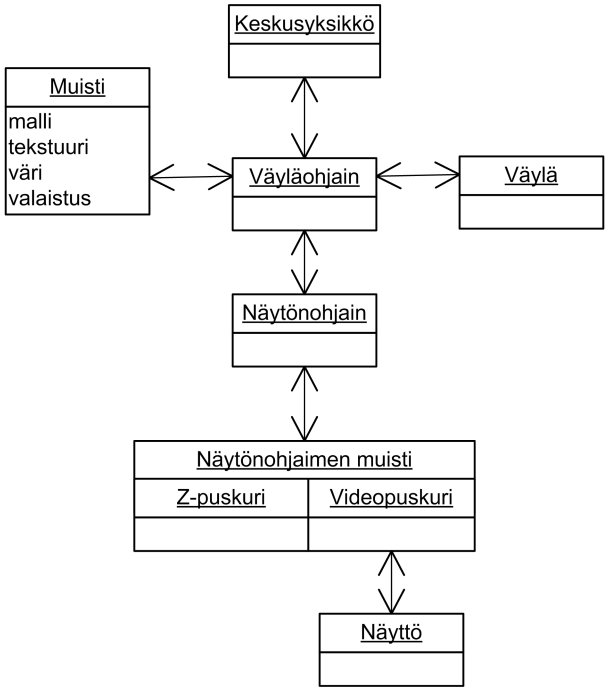
Geometriavaiheessa käsitellään koko maailmaa, joka sisältää kaikki sovellusvaihees-

sa luodut mallit. Geometriavaiheessa malleille suoritetaan sovellusvaiheessa määritellyt translaatiot, rotaatiot ja skaalaus. Tämän jälkeen mallit ovat käsiteltävissä maailmakoordinaatistossa, jossa suoritetaan kameramuunnos, jolla mallit saadaan muutettua katsojan koordinaatistoon. Katsojan koordinaatistossa malleille lasketaan valaistus, muodostetaan perspektiivi ja suoritetaan näkyvyystasojen normalisointi. Tämän jälkeen kuva muutetaan ikkunakoordinaatistoon rasterointia varten. Geometriavaiheessa valaistusten laskenta ja kiinteät muunnokset koordinaatistojen välillä voidaan laskea verteksisävyttimillä. Verteksisävytin suoritetaan jokaiselle kulmapisteelle eli verteksille.

Rasterointivaihe tuottaa lopullisen kaksiulotteisen pikseleistä koostuvan rasterikuvan. Geometriavaiheen jälkeen malli on muunnettu ikkunakoordinaatistoon, mutta mallin pisteet ovat edelleen kolmiulotteisia, eli niillä on myös syvyys. Rasterointivaiheessa mallista poistetaan monikulmiot ja malli kuvataan pikseleinä. Pikselit teksturoidaan ja niihin lisätään mahdollinen sumu ja niille määritetään väri, syvyys- ja siluettipuskurit sekä tehdään mahdollisesti syvyys- ja alphatesti. Syvyys- ja alphatestin perusteella määritetään, mitkä pikselit piirretään videopuskuriin ja tarvitseeko päällekkäin menevien pikselien värejä sekoittaa vai määrääkö päällimmäinen pikseli värin. Rasterointivaiheessa pikseleiden värin määritys tehdään pikselisävyttimien avulla, kuten valaistus geometriavaiheessa verteksisävyttimillä. Pikselisävyttimillä voidaan myös tarkentaa verteksisävyttimessä laskettua valaistusta ja tehdä muita efektejä kappaleen pintaan. Pikselisävytin suoritetaan jokaiselle pikselille. Rasterointivaiheen jälkeen kuva piirretään näytölle.

Kuvassa 3.1 on esitetty laitteisto, jonka läpi piirtoliukuhihna kulkee. Sovellusvaihe suoritetaan suorittimella, ja se tallettaa tietoa muistiin malleista, väreistä, tekstuureista ja muusta vastaavasta datasta. Sovellusvaiheen data siirretään väyläohjaimen avulla näytönohjaimelle, jossa suoritetaan geometriavaihe ja rasterointivaihe.

Geometriavaihe ja rasterointivaihe sisältävät paljon toimenpiteitä, joihin nimenomaan näytönohjain on erikoistunut (esim. geometriavaiheen matriisilaskut). Tämän takia geometriavaihe ja rasterointivaihe kannattaa pyrkiä suorittamaan aina näytönohjaimella, jos se vain on mahdollista. Sovellusvaiheessa puolestaan lasketaan usein kappaleiden liikettä, animaatioita ja fysikaalisia malleja. Peruslaskutoimitukset on usein nopeita suorittaa tietokoneen suorittimella, joten ne kannattaa suorittaa siellä.



Kuva 3.1: Tietokoneen suoritin ja näytönohjain [14]

3.2 3D-grafiikka selaimessa

3D-grafiikaa voidaan esittää selaimessa monilla eri tavoilla. Tässä työssä perehdytään tekniikoihin, joilla web-selaimessa 3D-grafiikan esittäminen ei vaadi liitännäisiä, kuten *FlashPlayeriä*. Tällaisia tekniikoita ovat *CSS 3D*, *SVG*, *Canvas* ja *WebGL*.

CSS 3D tarjoaa CSS-elementeille 3D-muunnokset. Tämä mahdollistaa CSS-tyylejä käyttävien HTML-elementtien rotaatiot ja translaatiot 3D-avaruudessa. CSS-määrittelyä käytetään muun muassa web-sovellusten käyttöliittymien tyylien määrittämiseen ja CSS 3D tuo mukanaan mahdollisuuden näyttäviin 3D-avaruudessa animoituihin web-käyttöliittymiin. CSS 3D on tarkoitettu kaksiulotteisten elementtien animointiin 3D-avaruudessa, mutta sen avulla on mahdollista koostaa myös 3D-malleja ja maailmoja 2D-objekteja käyttäen. 3D-mallien ja maailmojen koostaminen 2D-kappaleista on kuitenkin työlästä. [15]

SVG on suunniteltu interaktiivisen ja dynaamisen 2D-vektorigrafiikan esittämiseen XML-kuvauskieltä käyttäen [16]. SVG-grafiikkaa voidaan animoida kuvaamalla haluttu animaatio XML:lla tai skripteillä. 3D-grafiikan esittäminen SVG:lla vaatii kohdassa 3.1 kuvatun 3D-liukuhihnan joidenkin osien toteuttamista itse. SVG formaattiin on toteutettu myös laajennus, joka toteuttaa 3D-transformaatiot SVG -elementille, kuten CSS 3D CSS tyylejä käyttäville HTML -elementeille [17].

Canvas on HTML5-standardissa määritetty elementti, joka tarjoaa piirtorajapinnan 2D-grafiikalle. Canvas on tarkoitettu dynaamiseen 2D-grafiikan, kuten peligrafiikan, piirtoon [3]. Canvaksen avulla on kuitenkin mahdollista piirtää myös 3D-grafiikkaa samalla

tavalla kuin SVG-formaatilla, eli toteuttamalla osan piirtoliukuhihnasta itse.

Canvas tarjoaa myös mahdollisuuden piirtää laitteistokiihdytettyä 3D-grafiikka **WebGL**-rajapinnan avulla. WebGL perustuu OpenGL ES 2.0 standardiin ja se on suunniteltu kolmiulotteisen värillisen animoidun grafiikan esittämiseen. WebGL käyttää suoraan näytönohjainta grafiikan piirrossa, mikä tekee siitä hyvin tehokkaan. [4]

Suurin osa selaimen sisäisistä tekniikoista esittää 3D-grafiikkaa on alunperin suunniteltu 2D-grafiikan esittämiseen, mutta sittemmin niitä on erilaisten laajennusten ja kirjastojen myötä alettu käyttää myös 3D-grafiikan piirtoon. Ainoa varsinaisesti 3D-grafiikan esittämistä varten suunniteltu tekniikka on WebGL, ja siksi se on valittu tämän diplomityön teknisenä kontribuutiona rakennetun kirjaston pohjaksi.

3.3 WebGL

WebGL on OpenGL ES 2.0 -standardia mukaileva grafiikkaohjelmointirajapinta, joka mahdollistaa 3D-grafiikan esittämisen selaimessa HTML5 canvas-elementtiä käyttäen. WebGL toimii web-selaimessa ilman liitännäisiä. WebGL mahdollistaa esimerkiksi tietokonepeleistä tutun näyttävän 3D-grafiikan käyttämisen web-sovelluksissa. [4]

OpenGL ES 2.0 on vapaasti käytettävissä oleva alustariippumaton ohjelmistorajapinta grafiikkalaitteistolle, jota käyttäen ohjelmoija voi määrittää objekteja ja muita grafiikan piirtoon liittyviä ominaisuuksia. OpenGL ES on suunniteltu erityisesti sulautettuja järjestelmiä, esimerkiksi matkapuhelimia ja pelikonsoleita, ajatellen. OpenGL ES on kehitetty OpenGL 2.0 -standardista, joka on suunniteltu työpöytäkäytössä olevia tietokoneita varten. [18]

OpenGL ES 2.0 eroaa OpenGL 2.0:sta jonkin verran. OpenGL 2.0:ssa on esimerkiksi mahdollista käyttää niin sanottua kiinteää liukuhihnaa. Tämä tarkoittaa käytännössä sitä, että OpenGL 2.0 toteuttaa edellä kuvatun piirtoliukuhihnan geometriavaiheen ja rasterointivaiheen aina sävyttimistä lähtien. Kiinteää liukuhihnaa käyttäessä ohjelmoija pystyy muun muassa valitsemaan valmiiksi toteutetuista valaistusmalleista ja värityksistä käyttöön haluamansa. OpenGL ES 2.0 -versiossa ja myös OpenGL:n uudemmissa versioissa versiosta 3.1 lähtien kiinteä liukuhihna on poistettu. Kiinteän liukuhihnan poistaminen tarkoittaa sitä, että ohjelmoijan käytössä ei ole enää OpenGL 2.0:sta tuttuja valaistusmalleja ja värityksiä. Ilman kiinteää liukuhihnaa kehittäjä joutuu toteuttamaan itse pikseli- ja verteksisävyttimet käyttäen GLSL-sävytinkieltä. [18]

Kiinteän liukuhihnan puuttuminen tekee grafiikkaohjelmoinnista työläämpää, koska se siirtää enemmän asioita sovellusohjelmoijan tehtäväksi. OpenGL ES 2.0 onkin matalan tason grafiikkaohjelmointirajapinta, jonka lähtökohtana on tehokas toiminta myös sulautetuissa järjestelmissä. Kuten OpenGL ES 2.0:ssa, myöskään WebGL:ssä ei ole tukea kiinteälle liukuhihnalle. WebGL on OpenGL ES 2.0:n tavoin matalan tason API.

WebGL toimii selaimessa ilman liitännäisiä, mutta se käyttää selainta ajavan koneen näytönohjainta grafiikan piirtoon. Tämä asettaa vaatimuksia WebGL-sovelluksen

käyttäjän laitteistolle. Toimiakseen WebGL tarvitsee tarpeeksi uuden käyttöjärjestelmän: Windows-käyttöjärjestelmistä vähintään Windows XP:n ja MacOS X-käyttöjärjestelmistä vähintään version 10.6. Myös näytönohjaimessa täytyy olla ajantasainen OpenGL ES 2.0-ajuri, jotta WebGL sovellus toimisi. [19]

Mac OS X- ja Linux-pohjaisissa järjestelmissä pääasiallinen 3D-grafiikan ohjelmointirajapinta on OpenGL. Windows-pohjaisissa järjestelmissä puolestaan pääasiallinen rajapinta on Microsoftin DirectX rajapintakokoelmaan kuuluva Direct3D. Näin ollen kaikille näytönohjaimille Windows-ympäristöön ei ole saatavilla OpenGL-ajureita, jotka olisivat yhteensopivat OpenGL ES 2.0:n kanssa. OpenGL-ajurien puuttumisen takia edes kaikki teknisiltä vaatimuksiltaan muuten riittävät näytönohjaimet eivät ole yhteensopivia WebGL:n kanssa Windows ympäristöissä. Tätä ongelmaa on lähdetty ratkaisemaan muun muassa *ANGLE*-projektin (*Almost Native Graphics Layer Engine*) kautta. Projektin tarkoitus on toteuttaa välikerros, joka mahdollistaa OpenGL ES 2.0 -kutsujen välittämisen DirectX 9.0c -rajapinnalle. Tämä mahdollistaa WebGL:n käyttämisen Windows-ympäristössä myös näytönohjaimilla, joille ei ole sopivia OpenGL-ajureita. [20]

Microsoft on ilmoittanut, että tietoturvaongelmien takia WebGL-tukea ei toistaiseksi tule Internet Exploreriin. Kuten jo aikaisemmin on todettu, WebGL käyttää grafiikan piirtämiseen sovelluksen käyttäjän näytönohjainta, ja tämä mahdollistaa webin kautta tapahtuvat hyökkäykset näytönohjainta kohtaan, esimerkiksi näytönohjaimen ajurin heikkoutta hyödyntäen. [21]

Google ja Mozilla ovat puolestaan päättäneet pitää WebGL-tuen selaimissaan. Google ja Mozilla pitävät listaa näytönohjaimien ajureista, joista on raportoitu heikkouksia tai virheellistä toimintaa selaimien WebGL toteutuksen kanssa. Näitä listoja kutsutaan mustiksi listoiksi (engl. *blacklists*), ja WebGL on selaimen puolelta automaattisesti estetty, jos koneen näytönohjaimen ajuri on mustalla listalla. [19]

Koska vielä tällä hetkellä WebGL-sovelluksien saavutettavuus on melko pieni, useimmista WebGL-sovelluksista on toteutettu myös toinen versio esimerkiksi HTML5 canvaksen 2D-piirtorajapintaa käyttäen. Näin selain voi valita canvas-sovelluksen, jos WebGL-tukea ei ole. Google on lisäksi kehittänyt *SwiftShader*-työkalun, joka mahdollistaa WebGL:n käyttämisen silloinkin, kun näytönohjaimella ei ole tukea WebGL:lle tai kun näytönohjain tai sen ajuri on mustalla listalla. SwiftShader käsittelee WebGL-kutsut, jotka normaalisti menisivät näytönohjaimelle. SwiftShader on saatavilla Google Chrome-selaimeen versiosta 18 eteenpäin, ja se toimii ainoastaan Windows-käyttöjärjestelmässä. SwiftShader suorittaa piirron ohjelmallisesti keskusyksikön laskentatehoa käyttäen, kun taas WebGL suorittaa sen käyttäen näytönohjainta. Ohjelmallinen piirto näytönohjaimella suoritetun piirron sijaan toimii hyvin silloin, kun ohjelma ei ole graafisesti kovin intensiivinen. Vastaavasti esimerkiksi pelissä, jossa on paljon polygoneja, ei ohjelmallinen piirto yleensä pysty samaan tehokkuudessa kuin näytönohjaimella suoritettu piirto. [22]

Vaikka WebGL:n saavutettavuuteen liittyy vielä toistaiseksi edellä esiteltyjä ongel-

mia, on WebGL selaimen 3D-grafiikan toteutustavoista ainoa, joka on alun perin suunniteltu 3D-grafiikan esittämiseen. Näin ollen WebGL on luonnollinen valinta graafisesti intensiivisiin 3D-web-sovelluksiin. WebGL mahdollistaa web-sovelluksille sen, mikä on ollut mahdollista työpöytäsovelluksille jo pitkään: näyttävän laitteistokiihdytetyn 3D-grafiikan.

3.4 WebGL-kirjastot

Kuten kohdassa 3.3 todettiin, WebGL on matalan tason rajapinta, joka jättää sovellusohjelmoijan toteutettavaksi suuren osan piirtoliukuhihnaa. WebGL-sovellusten kehittämistä helpottamaan on kehitetty lukuisia erilaisia kirjastoja. Suurin osa kirjastoista on harrasteprojekteja tai saanut alkunsa tutkimusprojektista. Tässä kohdassa esiteltävät kirjastot ovat pitkälle kehitettyjä 3D-moottoreita, jotka tarjoavat korkean tason rajapinnat perusmuotojen toteuttamiseen, 3D-mallien lataamiseen, animointiin, teksturointiin, materiaaleihin ja valaistukseen.

Three.js[1] on Mr.doob nimimerkkiä käyttävän henkilön harrasteprojektina kehittämä 3D-moottori. Three.js kehitettiin alunperin canvaksen 2D-piirtorajapintaa käyttäväksi 3D-moottoriksi, mutta sittemmin moottoriin tehtiin myös WebGL-tuki. Tällä hetkellä three.js on yksi nopeimmin kehittyvistä ja paljon käytetyistä WebGL 3D-moottoreista. Kirjaston kehitystä seuraa tällä hetkellä github-palvelussa[2] 7800 käyttäjää. Three.js:ään voidaan tuoda resursseja, kuten 3D-malleja, muun muassa *Blender*- ja *3ds Max*[3] -mallinnusohjelmista. Three.js:ssä 3D-maailma muodostuu erillaisista resursseista, kuten valoista ja 3D-objekteista, jotka muodostavat hierarkisen kokonaisuuden. Three.js:llä on tehty paljon esimerkkejä ja demonstraatioita, jotka auttavat käyttäjää pääsemään alkuun kirjaston kanssa. Kirjaston dokumentaatio muilta osin on kuitenkin olematon. Kirjaston ympärille on muodostunut myös yhteisö, joka kehittää kirjastoa eteenpäin ja kirjoittaa blogisarjaa *Learning three.js*[4], joka esittelee kirjaston ominaisuuksien käyttöä perusteista alkaen. Three.js on saanut paljon näkyvyyttä erilaisissa WebGL-blogeissa, kuten *Learning WebGL* blogin *WebGL around the net* -osuudessa[5]. Näkyvyys on varmasti vaikuttanut siihen, että three.js:stä on tullut suosittu harrastajien keskuudessa. Myös tässä työssä toteutettu esimerkkisovellus käyttää Three.js 3D-moottoria. [23] [24]

GLGE: WebGL for the lazy[6] on Paul Bruntin harrastepohjalta kehittämä 3D-moottori. GLGE oli vielä kesällä 2010 yksi aktiivisimmin kehitetyistä WebGL -kirjastoista. Tällä hetkellä kirjaston kehitystä seuraa github-palvelussa noin 300 käyttäjää [25]. GLGE on hyvin dokumentoitu 3D-moottori, joka käyttää näkymien ja 3D-objektien

[1] three.js, http://github.com/mrdoob/three.js/
[2] github, http://github.com/
[3] 3ds Max, http://usa.autodesk.com/3ds-max/
[4] Learning three.js blogi, http://learningthreejs.com/
[5] Learning WebGL blogi, http://learningwebgl.com
[6] GLGE, http://www.glge.org/

määrittelyyn XML-kuvauskieltä. GLGE-kirjaston kanssa on mahdollista käyttää suoraan *Collada*-muodossa olevia 3D-malleja. GLGE-kirjaston vahvuutena on ehdottomasti sen kattava dokumentointi. Monet WebGL-kirjastot nojaavat dokumentoinnin osalta pelkästään esimerkkisovelluksiin, jotka eivät esittele kirjaston käyttöä kokonaisvaltaisesti. [24]

SceneJS[7] on tutkimusprojektista alkunsa saanut kirjasto. SceneJS koostaa 3D-näkymän puumaisesti erilaisista solmuista (engl. *node*). Solmu voi olla valonlähde, 3D-objekti, geometria, materiaali tai vaikka kamera. Kirjasto tarjoaa JSON-pohjaisen rajapinnan solmujen käsittelyyn. JSON on JavaScript pohjaiseen tiedonvaihtoon kehitetty standardi, joka määrittää mallin, jolla kuvataan avain–arvo-pareja. SceneJS tukee myös Collada-formaatissa olevia 3D-malleja. Kirjaston kehitystä github-palvelussa seuraa tällä hetkellä noin 200 käyttäjää [26]. [24]

[7] SceneJS, http://scenejs.org/

4 3D-KÄYTTÖLIITTYMÄKOMPONENTTI-KIRJASTON TOTEUTUS

Diplomityön teknisenä kontribuutiona toteutetun 3D-käyttöliittymäkomponenttikirjaston tarkoituksena on helpottaa interaktiivisten kolmiulotteisten web-sovellusten toteuttamista nostamalla abstraktiotasoa 3D-moottorin primitiiveistä käyttöliittymäkomponentteihin. Kirjasto koostuu kahdesta osasta: kirjaston rungosta ja sovitinosasta, joka on 3D-moottorin ja rungon välissä. Kirjasto on toteutettu kokonaan JavaScript-ohjelmointikielellä. Työn puitteissa toteutettiin kirjaston runko-osa ja sovitinrajapinta three.js 3D-moottorille.

Tässä luvussa käydään läpi kirjaston toteutukseen liittyviä suunnitteluperiaatteita ja ongelmia. Luvussa esitellään myös kirjaston rakenne ja sen käyttö.

4.1 Riippumattomuus 3D-moottorista

Käyttöliittymäkomponenttikirjaston yksi suunnitteluperiaate on ollut riippumattomuus käytettävästä 3D-moottorista. Käyttöliittymäkomponenttikirjasto kuitenkin tarvitsee 3D-moottorin palveluita muun muassa käyttöliittymän piirtämiseen ja käyttöliittymäkomponenttien esitystavan määrittämiseen. Näin ollen käyttöliittymäkomponenttikirjasto ei voi koskaan olla täysin riippumaton käytettävästä 3D-moottorista.

3D-moottorin ja komponenttikirjaston välille voidaan rakentaa sovitin. Sovitin toimii välikappaleena käyttöliittymäkomponenttikirjaston ja 3D-moottorin välillä. Sen tehtävänä on tarjota käyttöliittymäkomponenttikirjastolle rajapinta, jonka kautta se voi käyttää 3D-moottoria. Näin käyttöliittymäkomponenttikirjastoa ei tarvitse toteuttaa käyttäen jotain tiettyä 3D-moottoria. Kirjaston toteutus voidaan nyt jakaa kahteen osaan: runko-osaan, joka on riippumaton käytettävästä 3D-moottorista, ja sovittinosaan, joka on kirjaston 3D-moottorista riippuva osa.

3D-moottorin ja käyttöliittymäkomponenttikirjaston välinen sovitinrajapinta voitaisiin toteuttaa siten, että kaikki palvelut, joita käyttöliittymäkomponenttikirjasto tarvitsee 3D-moottorilta, toteutetaan käyttäen olemassaolevan 3D-moottorin, eli oletusmoottorin, rajapintaa. Käyttöliittymäkomponenttikirjaston ja oletusmoottorin välille ei tällöin tarvita sovitinta, mutta sovitinten kirjoittaminen muille 3D-moottoreille voisi olla hyvin monimutkaista tai jopa mahdotonta, koska tällä hetkellä WebGL 3D -moottorit tarjoavat palveluita hyvin vaihtelevilla abstraktiotasoilla ja niiden rajapinnat poikkeavat toisistaan joiltain

osin hyvin paljon.

Suurimmat tätä työtä toteuttaessa havaitut erot 3D-moottoreiden rajapinnoissa ovat törmäystarkastelussa. Törmäystarkastelua tarvitaan käyttöliittymäkomponenttikirjastossa hiiren tapahtumien käsittelyyn. Osa 3D-moottoreista tarjoaa työkaluja törmäystarkasteluun hyvinkin korkealla abstraktiotasolla ja toiset eivät ollenkaan. Lisäksi kahden eri kirjaston tarjoamat rajapinnat törmäystarkasteluun ovat hyvin erilaiset. Törmäystarkastelu on yksi niistä palveluista, johon sovittimen kirjoittaminen voisi olla hyvin hankalaa ellei jopa mahdotonta, jos törmäystarkastelu olisi suunniteltu tietyn 3D-moottorin mukaan. Törmäystarkastelu voidaan toteuttaa takaisinkutsun avulla, jolloin käyttöliittymäkomponenttikirjaston rungon ei tarvitse ottaa kantaa siihen, millaiset työkalut ja rajapinnat 3D-moottori tarjoaa. Törmäystarkastelun toteutus voidaan määritellä kirjaston sovitinosassa ja kirjaston runko-osa kutsuu sitä tarvittaessa.

3D-moottorit eroavat törmäystarkastelun lisäksi myös siinä, miten ne toteuttavat resurssien, kuten 3D-objektien, luomisen. 3D-moottori GLGE luo 3D-objektit XML-tiedoston pohjalta, kun taas SceneJS 3D-moottorin kaikki resurssit, myös 3D-objektit, määritellään JSON-muodossa. 3D-moottorissa three.js 3D-objektit luodaan kutsumalla kirjastoon määriteltyä 3D-objektin rakentajafunktiota. Myös 3D-moottorien tarjoamat perusmuodot ja palvelut 3D-mallien lataamiseen vaihtelevat huomattavasti.

Edellä esitetyt seikat aiheuttavat sen, että ainoa yhteinen asia, mitä 3D-moottorien tarjoamilla 3D-objekteilla on, on se, että niitä voi liikuttaa, pyörittää, piilottaa ja poistaa kirjaston tarjoaman rajapinnan avulla. Koska kirjaston runko-osan on tarkoitus olla riippumaton 3D-moottorin palveluista, esimerkiksi käyttöliittymäkomponenttien, joiden esitystapa (muoto, väri, koko, jne.) on ennalta määritelty, toteuttaminen kirjaston runko-osaan on käytännössä mahdotonta.

Tässä työssä toteutettu käyttöliittymäkomponenttikirjasto hoitaa asian siten, että kirjaston runko-osa ei itse luo instansseja käyttöliittymäkomponenttien 3D-objekteista. Esitystapa liitetään komponenttiin asettajametodin avulla. Käyttöliittymäkomponentin esitystapa, 3D-objekti, voidaan määrittää sovitinosassa tai kirjaston käyttäjä voi luoda sen itse käyttäen 3D-moottorin palveluita.

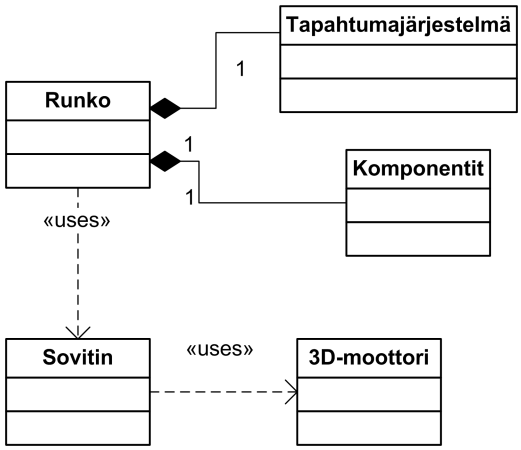
Tässä työssä toteutetun kirjaston rungon ja 3D-moottorin välinen rajapinta päätettiin toteuttaa siten, että se on mahdollisimman lähellä 3D-moottorien three.js ja SceneJS rajapintoja. Kummankin 3D-moottorin käyttöä varten täytyy kuitenkin tehdä erillinen sovitin, johon on toteutettu törmäystarkastelufunktio ja tarvittava yksinkertainen sovitinrajapinta. Kirjaston toteutuksen yhteydessä toteutettiin sovitin three.js 3D-moottorille. Toteutettua sovitinta käsitellään tarkemmin kohdassa 4.4.

4.2 Kirjaston rakenne

Käyttöliittymäkomponenttikirjasto koostuu runko- ja sovitinosasta. Käyttöliittymäkirjaston runko-osa sisältää yleisten käyttäliittymäkomponenttien toteutukset ja tapahtumajär-

jestelmän. Sovitin puolestaan tarjoaa 3D-moottorin palvelut kirjaston runko-osalle. Sovitinosaan voidaan toteuttaa myös 3D-moottorista riippuvia lisäpalveluita, kuten esitystavallisia käyttöliittymäkomponentteja.

Kuvassa 4.1 on esitetty kirjaston rakenne korkealla tasolla. Kirjaston runko-osa sisältää tapahtumajärjestelmän ja käyttöliittymäkomponenttien toteutuksen. Runko käyttää sovitinta, joka tarjoaa rungolle 3D-moottorin palvelut.



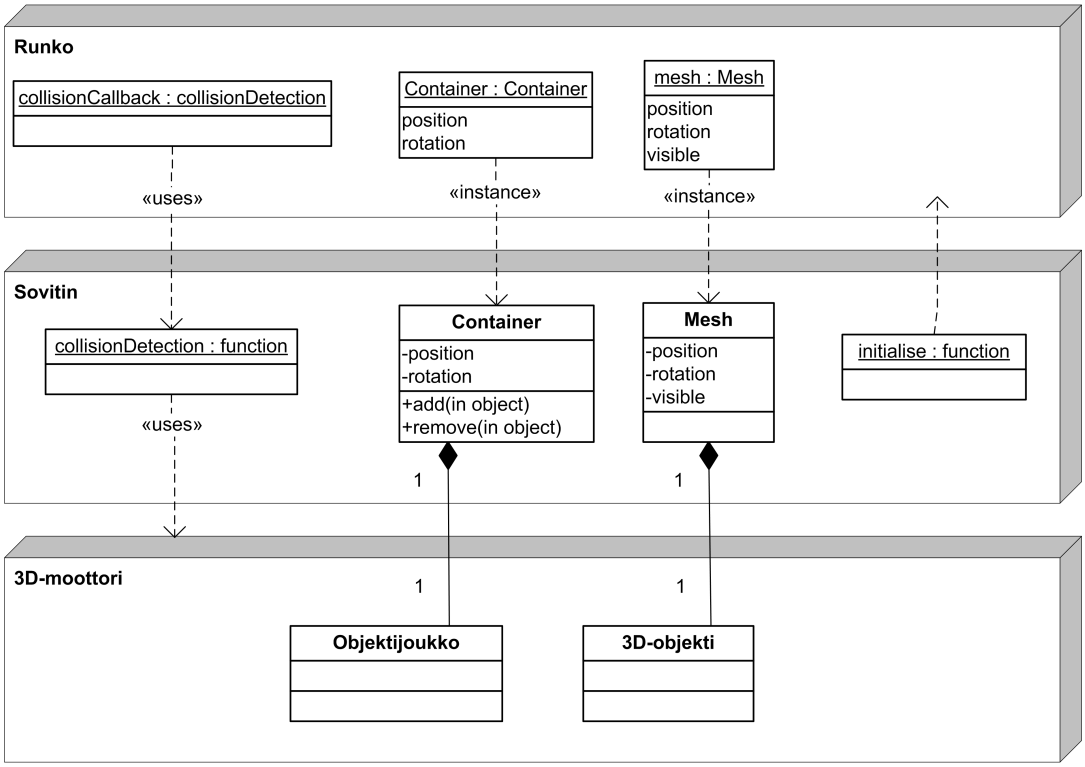
Kuva 4.1: Periaatekuva kirjaston rungon, sovittimen ja 3D-moottorin suhteesta

4.2.1 Kirjaston ja 3D-moottorin välinen rajapinta

Käyttöliittymäkomponenttikirjasto tarvitsee 3D-moottorilta joitain palveluita esimerkiksi komponenttien näkyvyyden ja transformaatioiden hallitsemiseen. Kuvassa 4.2 on esitetty minkälaisia abstraktioita ja palveluita kirjasto tarvitsee 3D-moottorilta, ja minkälaisten rajapintojen läpi käyttöliittymäkomponenttikirjaston runko käyttää 3D-moottorin objekteja.

Käyttöliittymäkomponenttikirjaston rungon tarvitsemia 3D-moottorista riippuvaisia objekteja ovat *mesh*, eli käyttöliittymäkomponentin 3D-objekti, ja *container*, eli säiliö, jolla voidaan yhdistää monta yksittäistä 3D-objektia ja säiliöitä yhdeksi objektijoukoksi. Käyttöliittymäkomponenttikirjasto tarvitsee objektijoukolta metodit *add(object)* ja *remove(object)*, joissa *object* on joko 3D-objekti tai objektijoukko.

Käyttöliittymäkomponentteja pitää pystyä myös liikuttamaan ja niiden näkyvyyttä pitää pystyä muuttamaan. 3D-objektilta ja objektijoukolta täytyy löytyä attribuutti *position*, joka kuvaa kappaleen keskipisteen koordinaatteja maailmakoordinaatistossa sekä attribuutti *rotation*, joka kuvaa kappaleen rotaatiota akseleiden x, y ja z suhteen. Kappaleiden rotaatioita ja sijaintia manipuloidaan muuttamalla arvoja *position.x, position.y* ja *position.z* sekä *rotation.x, rotation.y* ja *rotation.z*. Kun objektijoukkoa siirretään tai pyöritetään, kaikki sen lapset liikkuvat joukon keskipisteen suhteen saman verran samaan suuntaan. Lisäksi 3D-objektin täytyy sisältää attribuutti *visible*, joka määrittelee onko kappale



Kuva 4.2: Käyttöliittymäkomponenttikirjaston ja 3D-moottorin väliset rajapinnat

näkyvissä vai ei.

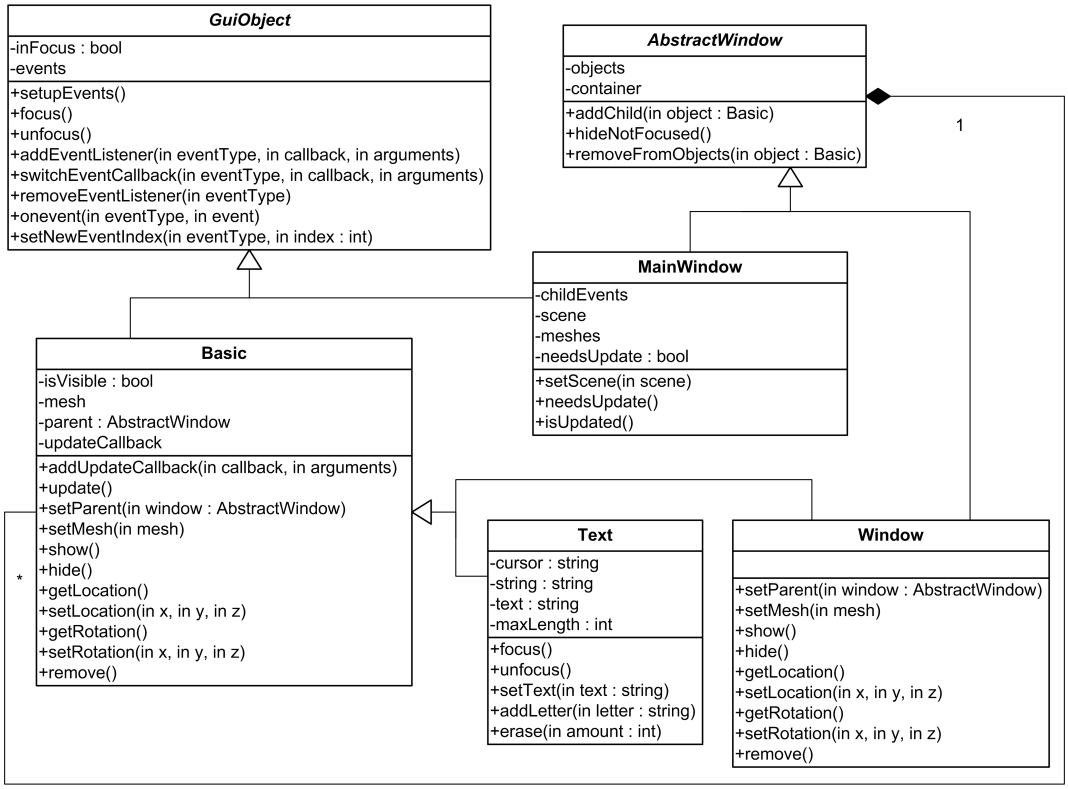
Objektijoukkoja on käytetty ikkunoiden toteutuksessa. Ikkunalle luodaan rakentajan suorituksen yhteydessä instanssi objektijoukosta, johon lisätään ikkunan 3D-objekti ja ikkunan lapsien 3D-objektit (lapsi-ikkunoiden tapauksessa lapsi-ikkunan objektijoukko). Ikkunan translaatiot ja rotaatiot tehdään objektijoukolle, jolloin ne periytyvät suoraan myös ikkunan lapsille. Kaikki kohdassa 3.4 käsitellyt 3D-moottorit toteuttavat objektijoukot jollain tavalla. Esimerkiksi three.js-kirjastossa joukon toteuttaa *Object3D* ja GLGE:ssä *Group*.

Kirjaston runko tarvitsee edellä esiteltyjen ominaisuuksien lisäksi myös törmäystarkastelufunktion, jonka avulla käyttöliittymäkomponenttikirjasto pystyy päättelemään hiiritapahtumien kohteen. Törmäystarkastelufunktio saa parametrinaan DOM-tapahtuman sekä törmäystarkastelufunktion mahdollisen parametriobjektin. Törmäystarkastelufunktio palauttaa käyttöliittymäkomponentin, johon hiiritapahtuma kohdistui, tai totuusarvon *false*, jos hiiritapahtuma ei osunut mihinkään näkyvissä olevaan komponenttiin.

Kirjasto otetaan käyttöön kutsumalla sovittimen alustusmetodia, joka hoitaa tarvittavan sovitinrajapinnan rekisteröimisen käyttöliittymäkomponenttikirjaston runko-osalle ja alustaa kirjaston runko-osan käyttövalmiiksi. Sovitinosa voi alustaa myös 3D-moottorin käyttövalmiiksi. Näin käyttöliittymäkomponenttikirjaston käyttäjän ei itse tarvitse huolehtia sovitinrajapintojen rekisteröimisestä kirjaston käyttöön.

4.2.2 Käyttöliittymäkomponentit

Toteutetun kirjaston runko-osa tarjoaa yksinkertaisia käyttöliittymäkomponentteja, joita käyttämällä ja yhdistelemällä käyttäjä voi itse koostaa monimutkaisempia komponentteja. Kirjaston rungon tarjoamia valmiskomponentteja ovat *MainWindow*, *Window*, *Text* ja *Basic*. Komponenttien tyyleihin tai esitystapaan kirjaston runko ei ota kantaa, mutta komponentit eroavat käyttötarkoituksiltaan. Kuvassa 4.3 on esitetty käyttöliittymäkomponenttikirjaston runko-osan tarjoamat komponentit luokkakaaviona.



Kuva 4.3: Luokkakaavioesitys kirjaston käyttöliittymäkomponenteista

Peruskomponentti, luokkakaaviossa Basic, on nimensä mukaisesti käyttöliittymäkomponentti, joka ei sisällä mitään erikoistunutta toimintaa. Peruskomponenttia voidaan käyttää sellaisenaan esimerkiksi nappina, valintaruutuna (engl. *checkbox*), kuvaelementtinä tai videoelementtinä. Basic periytyy luokasta *GuiObject*, kuten kaikki muutkin käyttöliittymäkomponenttikirjaston komponentit. Basic toimii kantaluokkana komponenteille Window ja Text, jotka näin ollen sisältävät kaikki samat ominaisuudet kuin peruskomponentti, mutta ovat erikoistuneempia.

Tekstikomponentti *Text* laajentaa peruskomponenttia tarjoamalla palveluita tekstin muokkaamiseen ja tallettamiseen. Tekstikomponentti on tarkoitettu käytettäväksi dynaamisen tekstin, kuten käyttäjän syötteen, esittämiseen.

Ikkunakomponentti *Window* on käyttöliittymäkomponentti, jonka lapseksi voidaan liittää muita Basic-tyyppisiä komponentteja. Ikkuna periytyy komponenteista *Basic* ja *Ab-*

stractWindow. Luokka *AbstractWindow* sisältää ikkunalle ja pääikkunalle yhteiset lapsikomponenttien hallitsemiseen liittyvät ominaisuudet. Ikkunakomponenttien toteutuksessa on käytetty rekursiokooste-suunnittelumallia.

Kirjaston runko-osa ei valmiiksi määrittele käyttöliittymäkomponenttien esitystapaa, eli 3D-maailman 3D-objektia. Ikkuna-, perus- ja tekstikomponentti sisältävät jäsenmuuttujan *mesh*, johon esitystapa, 3D-objekti, liitetään kutsumalla komponentin metodia *setMesh(object)*. Sovitin voi toteuttaa laajennettuja versioita käyttöliittymäkomponenteista, joissa esitystapa on ennalta määritelty. Jos sovittimen valmiista komponenteista ei löydy sovelluskehittäjälle mieluisaa, voi hän käyttää runko-osan komponentteja ja määrittää näille esitystavan 3D-moottorin avulla itse.

Pääikkuna *MainWindow* on hallintakomponentti, joka tarjoaa palveluita kaikkien käyttöliittymässä olevien komponenttien käsittelyyn. Lisäksi pääikkuna tarjoaa samat luokalta *AbstractWindow* perityt palvelut kuin ikkunakomponentti. Jokaisella sovelluksella on yksi pääikkunakomponentti, jonka käyttöliittymäkomponenttikirjasto luo kirjaston alustuksen yhteydessä.

Pääikkunakomponentti eroaa muista komponenteista myös siten, että sen toisena kantaluokkana toimii *GuiObject* eikä *Basic*. Pääikkuna ei näin ollen sisällä luokan Basic määrittämiä ominaisuuksia esimerkiksi esitystavan liittämiseen, komponentin näkyvyyden hallintaan eikä transformaatioihin. Pääikkunan esitystapana voidaan periaatteellisesti pitää 3D-moottorin 3D-maailmaa, jossa käyttöliittymä näytetään.

Kaikki esitellyt komponentit voivat ottaa vastaan näppäimistö- ja hiiritapahtumia. Vuorovaikutus komponenttien kanssa on toteutettu DOM-tapahtumien avulla, kuten perinteisissä web-käyttöliittymissä. Kuitenkin tapahtumia voi sitoa suoraan vain HTML-elementteihin. Kun sovelluksen interaktiiviset komponentit eivät ole HTML-elementtejä, ei tapahtumia pysty helposti sitomaan tiettyyn sovelluksen objektiin. Työssä rakennettu kirjasto määrittää yksinkertaisen rajapinnan, jonka avulla käyttäjä voi sitoa käyttöliittymäkomponentteihin takaisinkutsufunktiot yleisimmille DOM-tapahtumille. Kirjasto huolehtii siitä, että oikean käyttöliittymäkomponentin takaisinkutsufunktiota kutsutaan.

4.2.3 Omien komponenttien koostaminen

Kirjaston valmiina tarjoamia komponentteja on mahdollista laajentaa JavaScriptin prototyyppeihin perustuvan periytymisen ja koosteen avulla. JavaScriptin prototyyppeihin perustuva perintä mahdollistaa sen, että mitä tahansa objektia voidaan käyttää kantaluokkana. Kuitenkin komponenttikirjaston rungon tarjoamista objekteista vain *Basic*, *Window* ja *Text* on tarkoitettu käytettäväksi omien komponenttien rakentamiseen.

Periyttämisen avulla valmiisiin komponentteihin on mahdollista lisätä uusia ominaisuuksia ja uutta toiminnallisuutta sekä kirjoittaa yli perittyjä toimintoja. Koosteen avulla voidaan puolestaan luoda komponentteja, jotka sisältävät erilaisia valmiskomponentteja. Esimerkkinä koostetusta komponentista voisi olla vaikka ikkunakomponentti, jossa on

otsikkopalkki ja sulkunappi tai vaikka dialogi-ikkuna, joka koostuu tekstikentästä ja talletusnapista.

Käyttövalmiita komponentteja, joiden esitystapa on ennalta määrätty, voidaan toteuttaa samaan tapaan kuin esitystavasta riippumattomia komponentteja. Tällaiset komponentit ovat yleensä sovitinkohtaisia, eli ne ovat suoraan riippuvaisia käytettävästä 3D-moottorista, kuten kohdassa 4.1 todettiin. Esitystyyliltään valmiit komponentit kuitenkin helpottavat huomattavasti kirjaston käyttäjän työtä. Käyttäjä voi valita käyttöliittymäänsä sopivan näköiset komponentit, eikä hänen tarvitse huolehtia itse esitystavan määrittelemisestä.

Myös sovittimeen toteutettujen komponenttien tulee periytyä jostain kirjaston runko-osan komponentista. Näin voidaan varmistaa se, että sekä sovittimen että rungon komponenttien rajapinta pysyy yhtenäisenä. Komponenttien rajapintojen yhtenäisyys mahdollistaa esimerkiksi sen, että käyttäjä voi halutessaan rakentaa omia komponentteja laajentamalla ja koostamalla sovittimen tarjoamia komponentteja. Yhtenäiset rajapinnat mahdollistavat myös sovittimen vaihtaminen toiseen, joka tarjoaa samat palvelut, muuttamalla komponentteja käyttävää sovellusta mahdollisimman vähän.

4.3 Vuorovaikutus käyttäjän kanssa

Työssä rakennettu kirjasto määrittelee yhtenäisen rajapinnan hiiri- ja näppäimistötapahtumien sitomiseen komponentteihin. Käyttöliittymäkomponenttikirjasto saa DOM-tapahtuman selaimelta ja etsii tapahtumalle oikean käsittelijän.

Perinteisiä web-sovelluksia tehtäessä tapahtumakäsittelijöiden sitominen oikeisiin objekteihin on melko yksinkertaista. Hiiritapahtuma, kuten klikkaus, voidaan sitoa HTML-elementtiin. Kehittäjän ei tarvitse itse laskea, mitä elementtiä klikattiin, vaan selain osaa suoraan välittää tapahtuman oikealle tapahtumakäsittelijälle. Kolmiulotteisia WebGL-pohjaisia web-sovelluksia toteutettaessa tilanne on kuitenkin toinen.

WebGL:llä piiretty 3D-maailma esitetään HTML5 canvas-elementissä. 3D-maailmassa olevat 3D-objektit eivät ole HTML-elementtejä, joten niihin ei voi sitoa suoraan tapahtumakäsittelijöitä vaan tapahtumakäsittelijät on sidottava canvas-elementtiin. Canvas-elementtiin tulleesta tapahtumasta on pystyttävä päättelemään, mille 3D-maailman objektille se kuuluu vai kuuluko se millekään.

Toteutettu käyttöliittymäkomponenttikirjasto huolehtii tapahtumakäsittelystä oman tapahtumajärjestelmänsä avulla. Kun käyttöliittymäkomponenttiin sidotaan tapahtumakäsittelijä, talletetaan tapahtumakäsittelijäfunktio käyttöliittymäkomponenttiin ja komponentti talletetaan pääikkunan tapahtumatauluun oikean tapahtuman kohdalle. Kun tapahtuma tulee, kutsutaan käyttöliittymäkomponenttikirjaston keskitettyä tapahtumakäsittelijää, joka etsii pääikkunan tapahtumataulusta oikean komponentin, jonka tapahtumakäsittelijää kutsutaan. Jokaiselle tuetulle tapahtumalle on määritetty oma keskitetty käsittelijä, joka kutsuu tapahtuman tyypistä riippuen erilaisia apufunktioita oikean tapahtumakäsitte-

lijän löytämiseksi ja sen löydyttyä välittää tapahtuman sille. Kirjaston tukemat tapahtumat on listattu taulukossa 4.1.

Taulukko 4.1: Komponenttikirjaston tukemat DOM-tapahtumat

Tapahtuma	**Välitetään, kun**
onclick	hiiren nappia klikataan.
ondblclick	hiiren nappia tuplaklikataan.
onmousemove	hiirtä liikutetaan.
onmousedown	hiiren nappi painetaan alas.
onmouseup	hiiren nappi nousee ylös.
onmouseover	hiiren kursori liikkuu kappaleen päälle.
onmouseout	hiiren kursori liikkuu pois kappaleen päältä.
onkeydown	näppäimistön näppäin painetaan alas.
onkeyup	näppäimistön näppäin nousee ylös.
onkeypress	näppäimistön näppäintä painetaan.

Oikean käyttöliittymäkomponentin etsimiseen hiiritapahtuman tullessa kuuluu muun muassa törmäystarkastelun tekeminen, jonka tapahtumajärjestelmä hoitaa. Koska hiiritapahtumien koordinaatit ovat näytön koordinaatistossa, joudutaan ne ensin muuntamaan web-sivun koordinaatistoon ja siitä edelleen 3D-maailmakoordinaatistoon. Tämän jälkeen voidaan tehdä pisteen törmäystarkastelu maailmassa oleviin 3D-objekteihin. Kun törmäystarkastelu on suoritettu ja tiedetään, mille käyttöliittymäkomponentille tapahtuma kuuluu, kutsutaan komponentin omaa tapahtumakäsittelijää. Jos hiiritapahtuma ei osunut mihinkään näkyvään 3D-objektiin, tarkastetaan löytyykö pääikkunalta sopivaa käsittelijää ja jos sellainen löytyy, kutsutaan sitä.

Oikean komponentin etsiminen hiiritapahtuman tullessa voi olla raskas operaatio, jos käyttöliittymä sisältää paljon 3D-objekteja. Hiiren törmäystarkastelu oikean objektin löytämiseksi on optimoimattomana kertaluokassa $O(n)$ (jossa *n* on 3D-objektien määrä käyttöliittymässä), eli hiiri voi törmätä mihin tahansa käyttöliittymän kappaleeseen. Tapa, jolla törmäystarkastelu on toteutettu vaikuttaa paljon hiiritapahtumien käsittelyn tehokkuuteen. Kirjaston törmäystarkastelufunktio on toteutettu kirjaston sovittimeen, jolloin se on voitu optimoida käytettävälle 3D-moottorille. Käyttäjä voi halutessaan myös korvata sovittimen toteuttaman törmäystarkastelufunktion omalla funktiollaan.

Näppäimistötapahtumien käsittely eroaa hiiritapahtumista siten, että niiden välittämiseen ei tarvita törmäystarkastelun laskemista. Jos näppäimistötapahtumia vastaanottavia komponentteja on useita, täytyy kirjaston pystyä päättelemään, mille komponentille tapahtuma välitetään. Jokaisella käyttöliittymäkomponentilla on jäsenmuuttuja *inFocus*, joka sisältää tiedon siitä, onko komponentti valittuna vai ei. Näppäimistötapahtuma välitetään kaikille komponenteille, joilta löytyy kyseiseen tapahtumaan käsittelijä ja jotka ovat tapahtuman tullessa valittuna.

Niin sanottu turhien tapahtumien käsittely on pyritty kirjaston puolelta välttämään siten, että tapahtumaa aletaan kuunnella vasta, kun vähintään yhteen komponenttiin rekisteröidään tapahtumalle käsittelijä. Tapahtuman kuuntelu lopetetaan, kun viimeinen tapahtumaan rekisteröity tapahtumakäsittelijä poistetaan.

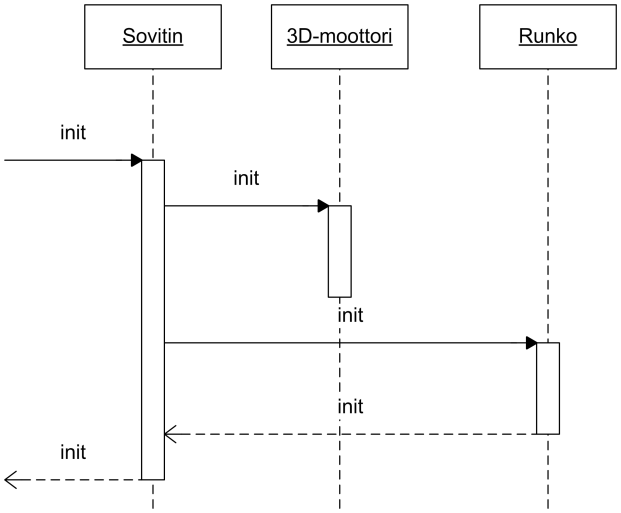
Tapahtumakäsittelijän liittäminen komponenttiin tapahtuu kutsumalla komponentin metodia *addEventlistener(eventType, callback, args)*, jossa *eventType* on tapahtuman tyyppi, *callback* tapahtumankäsittelijäfunktio ja *args* käsittelijäfunktiolle välitettävä parametriobjekti. Tapahtumakäsittelijän voi vaihtaa metodilla *switchEventCallback(eventType, callback, args)* ja poistaa metodilla *removeEventListener(eventType)*.

4.4 Sovitin three.js 3D-moottorille

Vaikka kirjaston toteutus on jaettu kahteen osaan, on sovitin kiinteä osa toteutettua käyttöliittymäkomponenttikirjastoa. Komponenttikirjasto on tarkoitettu käytettäväksi sovittimen kautta, jolloin käyttäjän ei itse tarvitse huolehtia sovitinrajapinnoista eikä törmäystarkastelun implementoimisesta. Lisäksi käyttäjällä on käytössään sovittimen tarjoamat tyylitellyt komponentit, jotka helpottavat käyttöliittymien rakentamista entisestään. Tämän työn puitteissa kirjastoon toteutettiin sovitin three.js 3D-moottorille. Sovitin on toteutettu three.js-kirjaston versiolle 49.

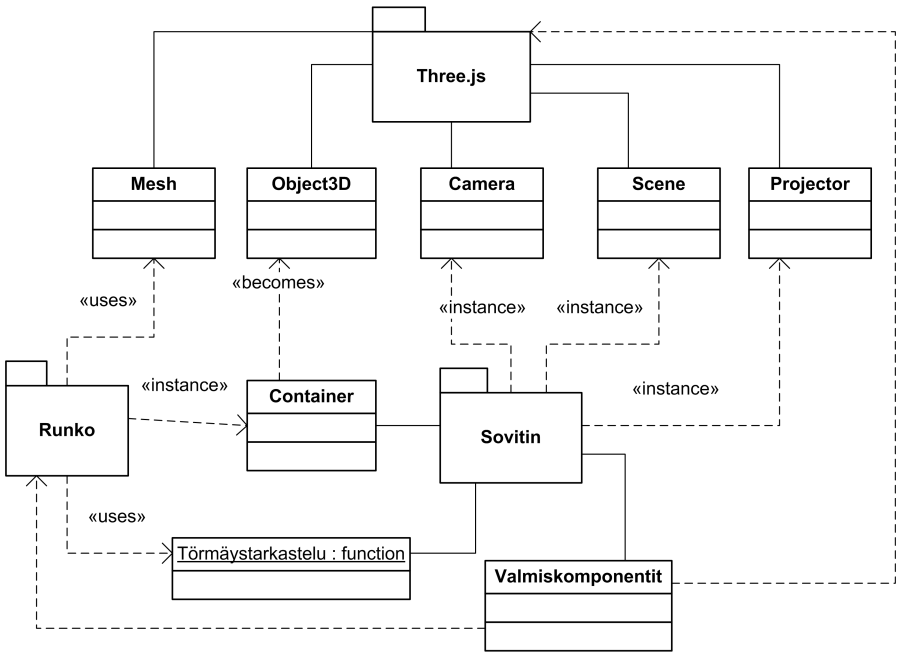
4.4.1 Yleinen toiminta

Sovittimen tekeminen three.js 3D-moottorin ja kirjaston rungon väliin on melko yksinkertaista. Three.js-kirjaston 3D-objektin ja objektijoukon rajapinnat ovat suoraa yhteensopivat komponenttikirjaston rungon rajapintojen kanssa. Varsinaista sovitinrajapintaa 3D-moottorin objektien ja kirjaston objektien välille ei siis tässä tapauksessa tarvita. Sovitin toteuttaa kuitenkin törmäystarkastelufunktion kirjaston käyttöön ja alustusfunktion, joka huolehtii sovittimen rekisteröimisestä kirjaston runko-osan käyttöön ja kirjaston runko-osan ja three.js 3D-moottorin alustamisesta. Käyttöliittymäkomponenttikirjasto otetaan käyttöön sovittimen kanssa kutsumalla sovittimen alustusfunktiota. Alustus palauttaa onnistuessaan käyttöliittymän pääikkunakomponentin. Kuvassa 4.4 on esitetty sekvenssikaavio kirjaston käyttöönotosta.



Kuva 4.4: Kirjaston alustus sekvenssikaaviona

Kuvassa 4.5 on esitetty sovittimen, kirjaston rungon ja three.js:n väliset riippuvuudet. Sovitin sisältää rajapinnan objektijoukolle (*Container*), jota käyttämällä kirjaston runko-osa voi luoda instansseja three.js:n objektijoukosta (*Object3D*). Sovitin luo alustuksen yhteydessä instanssit three.js:n tarjoamista objekteista *Camera*, *Scene* ja *Projector*, joita törmäystarkastelufunktio käyttää. Lisäksi sovitin tarjoaa valmiskomponentteja, jotka käyttävät kirjaston runko-osan palveluita ja three.js-kirjaston palveluita.



Kuva 4.5: Sovittimen, kirjaston rungon ja three.js:n riippuvuuskaavio

Kuten aiemmin todettiin, käyttöliittymäkomponenttikirjaston runko ei luo instanssia käyttöliittymäkomponenttiin liittyvästä 3D-objektista, vaan 3D-objekti liitetään komponenttiin metodilla *setMesh*. Three.js-kirjaston tarjoaman 3D-objektin (*Mesh*) rajapinta

on suoraan yhteensopiva käyttöliittymäkomponenttikirjaston rungon vaatiman rajapinnan kanssa. Tämän takia sovittimessa ei ole toteutettu rajapintaa, jonka avulla kirjaston käyttäjä voisi luoda 3D-objekteja. Jos kirjaston käyttäjä käyttää kirjaston runko-osan komponentteja, hänen tulee määritellä 3D-objekti suoraa three.js-moottoria käyttäen ja tämän jälkeen liittää 3D-objekti haluamaansa käyttöliittymäkomponenttiin. Sovittimen yhteyteen toteutetut valmiskomponentit kuitenkin abstrahoivat 3D-objektien luomisen käyttäjältä. Näiden tyyliteltyjen komponenttien esitystapa on ennalta määritelty, joten komponenttiin liittyvä 3D-objekti luodaan ja liitetään komponenttiin automaattisesti komponentin luomisen yhteydessä.

Three.js tarjoaa hyvät palvelut törmäystarkastelun, erityisesti hiiren törmäystarkastelun, toteuttamiseen. Three.js tarjoaa *projector*-objektin, jonka avulla voidaan suorittaa matriisioperaatioita, kuten projektioita ja takaisinprojektioita. *Projector* tarjoaa valmiina myös metodin *pickingRay*, jonka avulla voidaan luoda hiiren paikasta säde 3D-maailmaan. Metodin palauttamalla säteellä puolestaan on metodi, jonka avulla pystytään tarkastamaan säteen leikkaukset maailman objektien kanssa, tämä metodi on *intersectsObjects(objects)*, jossa *objects* on taulukko 3D-objekteista, joihin leikkausta tarkastellaan. Metodi *intersectsObjects* palauttaa taulukon objekteista, joiden kanssa säde leikkasi. Palautettu taulukko ei kuitenkaan sisällä pelkästään säteen kanssa leikanneita 3D-objekteja, vaan taulukossa olevat objektit sisältävät myös tiedot leikkauspisteen etäisyyden säteen lähtöpisteeseen ja maailmakoordinaatiston kohdan, jossa säde komponentin leikkasi.

Sovittimessa törmäystarkastelu on toteutettu edellä kuvattuja työkaluja käyttäen. Sovitin muuntaa hiirikoordinaatit näytön koordinaatistosta välille -1...1, ja antaa nämä *pickingRay*-metodille, joka muuntaa koordinaatit maailmakoordinaatistoon ja luo pisteestä lähtevän säteen. Säteen leikkauksia tarkastavalle metodille annetaan parametrina taulukko niiden käyttöliittymäkomponenttien 3D-objekteista, joilla on tapahtumakäsittelijä tapahtumalle, jonka kohdetta etsitään. Näin törmäystä ei tarvitse tarkastaa kaikkien käyttöliittymässä olevien 3D-objektien suhteen. Säde voi kuitenkin leikata monta käyttöliittymäkomponenttia. Oikean komponentin valinta tapahtuu valitsemalla säteen leikkaamista komponenteista lähimpänä säteen alkupistettä oleva näkyvä (3D-objektin attribuutin *visible* on oltava tosi) komponentti.

4.4.2 Valmiskomponentit

Sovittimeen on toteutettu valmiskomponentteja, joille on määritelty valmiit tyylit. Toteutetut komponentit laajentavat käyttöliittymäkomponenttikirjaston runko-osan komponentteja.

Otsikollinen ikkunakomponentti, *TitledWindow*, on yksi valmiiksi tyylitellyistä komponenteista. Komponentti koostuu ikkunakomponentista ja kahdesta peruskomponentista. Toista peruskomponenttia käytetään ikkunan otsikkona ja toista sulkunappina. Ikkunan

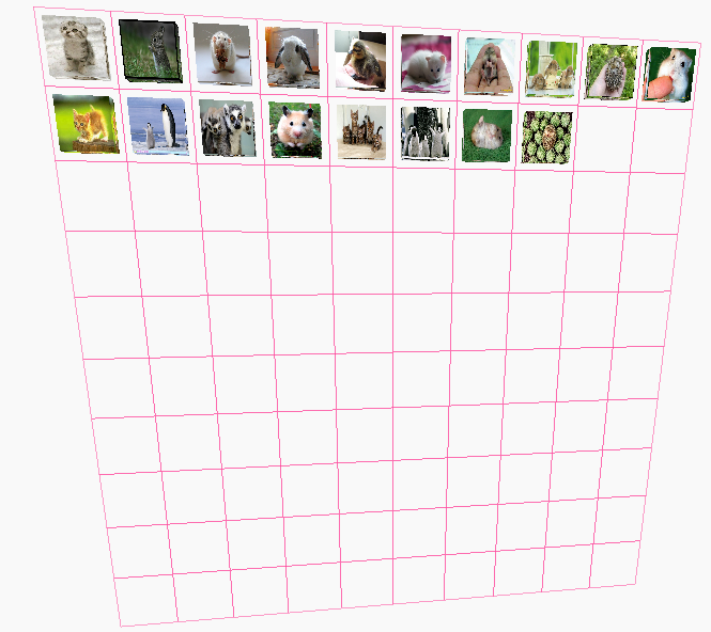
esitystapana on taso. Otsikkokomponentti on sijoitettu ikkunan yläreunaan ja sulkunappi otsikon viereen oikealle. Käyttäjä voi määrittää ikkunan leveyden ja korkeuden, ja vaihtaa ikkunan otsikkoa. Kuvassa 4.6 näkyy *TitledWindow*-komponentti, jonka 3D-objekti on teksturoitu HTML5 video-elementillä.

Kuva 4.6: Videoikkuna toteutettuna *TitledWindow*-komponentilla

GridWindow on ikkunakomponentti, jonka esitystapana on neliön muotoinen ruudukko. Ruudukkoikkunan lapseksi voi liittää *GridIcon*-komponentteja. *GridIcon* laajentaa komponenttia *Basic* ja sen esitystapa on kuutio, jonka materiaalin käyttäjä voi määrittää. Kun *GridIcon*-komponentti liitetään *GridWindow*-komponentin lapseksi, se asetetaan automaattisesti ruudukkoikkunaan seuraavaan vapaaseen ruutuun. *GridIconin* koko riippuu sen vanhemman koosta. *GridWindow*-komponentti soveltuu käytettäväksi esimerkiksi tiedostoselaimen esitystapana, jolloin ikkunaan liitetyt *GridIcon*-lapsikomponentit toimivat tiedostojen pikakuvakkeina. Kuvassa 4.7 on kuvakaappaus sovelluksesta, jossa on käytetty *GridWindow*- ja *GridIcon*-komponentteja.

GridWindow- ja *TitledWindow*-komponentteihin on toteutettu valmiiksi tapahtumakäsittelijät *onmousedown*, *onmousemove* ja *onmouseup* tapahtumille. *GridWindown* valmiit tapahtumakäsittelijät mahdollistavat komponentin pyörittämisen hiirellä x- ja y-akselien suhteen. *TitledWindown* käsittelijät puolestaan toteuttavat komponentille kontrollit, joiden avulla ikkunaa voidaan liikuttaa xy-tasossa raahamalla sitä hiirellä otsikkopalkista. Kontrolleiden toteuttamat paikanmuutokset, animointi, astuvat voimaan kun komponentin metodia *update()* kutsutaan.

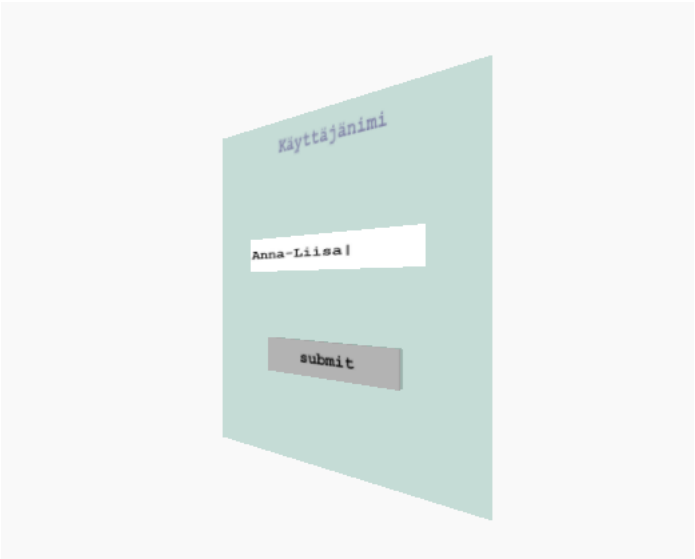
Sovitin tarjoaa myös kahdenlaisia dialogi-ikkunoita: *Dialog* ja *SelectDialog*. Komponenteista *Dialog* on tarkoitettu sellaisten dialogien esittämiseen, jossa käyttäjältä pyydetään syötteitä. Komponentin runkona toimii *Window*-komponentti, jonka esitystapa on



Kuva 4.7: Kuvaselain toteutettuna GridWindow- ja GridIcon-komponenttien avulla

läpikuultava taso, johon käyttäjä voi määrittää dialogin ohjetekstin. *Dialogiin* liittyy *Text*-komponentti, johon käyttäjän syöte luetaan ja *Basic*-komponentti, joka toimii nappina.

Dialog-komponentin tekstikentälle on määritetty tapahtumakäsittelijät näppäimistötapahtumille ja hiirenklikkaukselle. Hiirenklikkaus aktivoi tekstikentän ja aktivoituun tekstikenttään kirjoitettava teksti talletetaan komponenttiin. Käyttäjä toteuttaa itse tapahtumakäsittelijät komponentin muille osille, kuten napille. Napin hiirenklikkauskäsittelijässä voidaan esimerkiksi tarkastaa ja tallettaa käyttäjän tekstikenttään antama syöte ja piilottaa dialogi. Kuvassa 4.8 on *Dialog*-komponentti.



Kuva 4.8: Käyttäjän syötteen vastaanottaminen toteutettuna Dialog-komponentilla

SelectDialog on puolestaan tarkoitettu esittämään dialogeja, joissa käyttäjä valitsee toiminnon dialogissa esitetyistä vaihtoehdoista. *SelectDialog*-komponentin runkona toimii ikkunakomponentti. Dialogin vaihtoehtolista muodostuu *Basic*-komponenteista. Käyttäjä antaa *SelectDialogin* luomisen yhteydessä komponentin rakentajalle parametrina taulukon dialogissa esitettävistä valintavaihtoehdoista. Taulukon alkio, vaihtoehto, on objekti, joka sisältää tekstin, joka esitetään vaihtoehtoon liitettävässä *Basic*-komponentissa, sekä *onclick*-tapahtuman käsittelijäfunktion, joka suoritetaan, jos käyttäjä klikkaa hiirellä vaihtoehtoa. Kuvassa 4.9 on kuva *SelectDialog*-komponentista.

Kuva 4.9: SelectDialog-komponentti

Kaikki valmiskomponenttien sisältämät tekstit, kuten otsikot, valikkotekstit ja käyttäjän syöteteksti, on toteutettu käyttäen 3D-objektin tekstuurina canvas-elementtiä, jolle teksti on piirretty. Canvaksen käyttö tekstuurina on yksinkertaisin tapa toteuttaa dynaamista tekstiä WebGL:llä. WebGL ei itsessään tue mitenkään tekstinpiirtoa eivätkä monet 3D-moottoritkaan sisällä työkaluja dynaamisen tekstin esittämiseen.

4.5 Kirjaston käyttö

Kuten aiemmin todettiin, käyttöliittymäkomponenttikirjasto on tarkoitettu käytettäväksi sovittimen kanssa. Kirjaston alustus tapahtuu kutsumalla kirjaston sovittimen alustusfunktiota, joka palauttaa käyttöliittymän pääikkunakomponentin.

Ohjelmoija voi rakentaa haluamansa käyttöliittymän käyttäen sovittimen valmiita komponentteja. Jos käyttötarkoitukseen sopivia valmiskomponentteja ei ole, voi käyttäjä käyttää kirjaston runko-osan komponentteja ja määrittää komponenttien esitystavan itse. Myös monimutkaisempien komponenttien koostaminen on mahdollista ja sovitinosaan toteutettuja valmiskomponentteja voi käyttää hyväksi monimutkaisempia komponentteja rakennettaessa.

Jokaisella käyttöliittymäkomponentilla pääikkunaa lukuun ottamatta on aina vanhempi. Käyttöliittymäkomponentti voidaan lisätä ikkunakomponentin lapseksi ikkunakomponentin metodilla *addChild(object)*. Komponentti voidaan poistaa käyttöliittymästä kutsumalla komponentin metodia *remove()*. Jos komponentti halutaan poistaa kokonaan sovelluksesta, on tärkeää poistaa myös kaikki viitteet, jotka siihen liittyvät. Komponenttikirjasto huolehtii komponentin poiston yhteydessä kirjaston sisäisten viitteiden poistamisesta, mutta käyttäjän täytyy itse huolehtia oman sovelluksensa sisältämien viitteiden poistamisesta. Pääikkunakomponenttia ei voi poistaa metodilla *remove()* eikä sitä voi myöskään asettaa minkään komponentin lapseksi.

Käyttöliittymä muodostaa puumaisen rakenteen, jonka juurena on käyttöliittymän pääikkunakomponentti. Kaikki käyttöliittymään liittyvät komponentit on löydettävissä tämän komponentin kautta. Käyttöliittymäkomponenteista vain ikkunakomponenteilla voi olla lapsikomponentteja. Pääikkunakomponentteja käyttöliittymässä on aina vain yksi. Käyttäjän ei tarvitse huolehtia pääikkunakomponentin luomisesta, koska se luodaan kirjaston alustuksen yhteydessä. Kirjaston metodi *getMainWindow()* palauttaa viitteen kirjaston pääikkunakomponenttiin.

Käyttöliittymäkomponentit tarjoavat yhtenäisen rajapinnan komponenttien tapahtumakäsittelyyn, fokusointiin, transformaatioihin ja esitystavan liittämiseen. Lisäksi ikkunakomponentit tarjoavat metodeja, joiden avulla ikkunan lapsia voidaan hallita. Esimerkiksi ikkunakomponentin metodi *hideNotFocused()* piilottaa kaikki ikkunan lapsikomponentit, jotka eivät ole sillä hetkellä valittuna.

Kaikki *Basic*-komponentista periytyvät komponentit toteuttavat myös rajapinnan, jonka avulla komponenttiin voidaan lisätä päivitysfunktio. Päivitysfunkio on käyttäjän määrittämä ja siinä voidaan esimerkiksi animoida komponenttia tai päivittää komponentin tekstuuria. Päivitysfunktio lisätään komponentille *addUpdateCallback(function, parameters)*, jossa *function* on kutsuttava päivitysfunktio ja *parameters* päivitysfunktiolle parametrina annettava objekti. Komponentti kutsuu päivitystakaisinkutsufunktiota, kun sen metodia *update()* kutsutaan.

Esimerkkinä kirjaston käytöstä ohjelmassa 4.1 on esitetty kuvassa 4.7 olevan yksinkertaisen kuvaselaimen toteutus. Toteutuksessa luodaan *GridWindow*-komponentti, jonka korkeus ja leveys ovat 2000, ruutuja ikkunassa on 10 vaaka- ja pystyriveillä, ruudukon väri on vaaleanpunainen ja ikkuna käyttää komponentin toteuttamia hiirikontrolleja, joiden avulla ikkunaa voidaan käännellä x- ja y-akselien suhteen. Vertailun vuoksi liitteessä A on esitetty saman esimerkin toteutus pelkästään three.js 3D-moottorin palveluita käyttäen.

Ohjelma 4.1: Yksinkertaisen kuvaselaimen toteutus käyttöliittymäkomponenttikirjastoa käyttäen.

```
1  //Funktio, jota kutsutaan, kun HTML-dokumentti on latautunut.
2  var kuvaselain = function(){
3
4    //Taulukko kuvista, jotka esitetään kuvaselaimessa.
5    var pictures =
6      ["img/kitten1.jpg", "img/bunny1.jpg", "img/rat1.jpg", "img/bunny2.jpg"];
7
8    // Kutsutaan three.js välikerroksen alustusfunktiota, joka alustaa kirjaston
         käyttövalmiiksi. Parametrina annetaan canvaselementin haluttu koko, mikä on tässä
         tapauksessa selainikkunan koko. Lisäksi annetaan ruudun tyhjennysväri.
9    var mainWindow = THREEJS_WIDGET3D.init({canvasWidth : window.innerWidht,
10     canvasHeight : window.innerHeight,
11     clearColo : 0xf9f9f9});
12   //Siirretään 3D-moottorin kamera halutulle paikalle.
13   THREEJS_WIDGET3D.camera.position.z = 1600;
14
15   //Luodaan GridWindow-komponentti kuville.
16   var gridWindow = new THREEJS_WIDGET3D.GridWindow({width : 2000,
17     height : 2000,
18     density : 10,
19     color : 0x6B6B6B,
20     defaultControls : true});
21
22   //Lisätään komponentti käyttöliittymän pääikkunan lapseksi.
23   mainWindow.addChild(gridWindow);
24
25   //Luodaan kuvat ja asetetaan ne GridWindow-komponentin lapsiksi.
26   for (var i = 0; i < pictures.length; ++i){
27
28     //Haluttu tekstuurikuvan polku annetaan parametrina komponentin rakentajalle, samoin
           komponentin vanhempi.
29     var button = new THREEJS_WIDGET3D.GridIcon({picture : pictures[i], parent :
           gridWindow});
30   }
31
32   //Animointisilmukan toteutus.
33   var mainLoop = function(){
34     //Päivitetään ikkunakomponentti, jotta mahdolliset ikkunan rotaatiot tulisi voimaan.
35     gridWindow.update();
36     //Piirretään maailma.
37     THREEJS_WIDGET3D.render();
38   };
39
40   //Asetetaan intervalli (millisekuntteina), jonka välein animointisilmukkaa kutsutaan.
         Nyt funktiota kutsutaan 60 kertaa sekunnissa eli maksimi FPS on 60.
41   setInterval(mainLoop, 1000/60);
42 }
```

5 ESIMERKKISOVELLUS

Diplomityön teknisenä kontribuutiona rakennetun kirjaston lisäksi työn puitteissa toteutettiin esimerkkisovellus kirjastoa käyttäen. Esimerkkisovelluksena muokattiin Jari-Pekka Voutilaisen diplomityönä toteuttama *Lively3D*-ikkunointiympäristö [7] käyttämään tässä työssä toteutettua käyttöliittymäkomponenttikirjastoa.

5.1 Lively3D

Lively3D on ikkunointiympäristö, johon on mahdollista upottaa HTML5 canvas-elementteihin rakennettuja sovelluksia. Ikkunointiympäristö käyttää GLGE:tä 3D-maailman ja 3D-käyttöliittymän toteuttamiseen. Ikkunointiympäristö voidaan visualisoida monella eritapaa ja siihen on mahdollista toteuttaa omia visualisaatioita Lively3D:n tarjoaman rajapinnan avulla. [7]

5.1.1 Taustaa

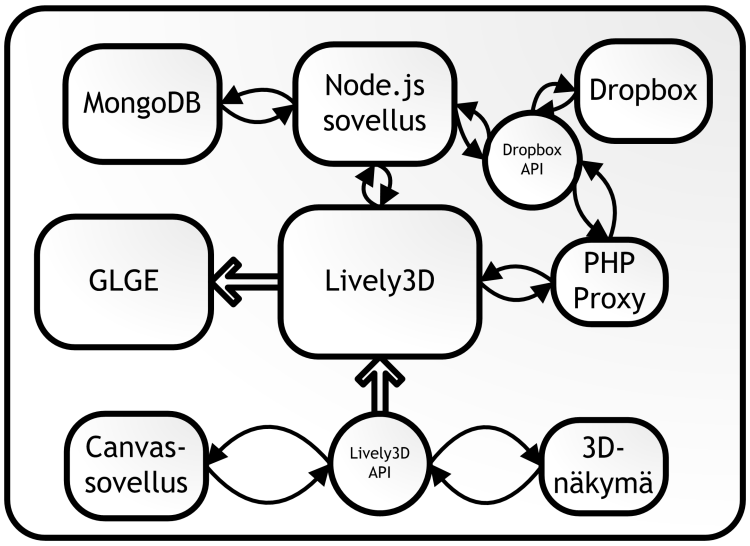
Lively3D on toteutettu uusien verkkoteknologioiden tuomien mahdollisuuksien ja rajoitteiden tutkimista varten. Lively3D toteuttaa ohjelmointirajapinnat, joiden avulla ympäristöön voidaan upottaa canvas-sovelluksia ja työpöydän visualisaatioita eli 3D-ympäristöjä. Sovellukset ja 3D-ympäristöt voidaan asentaa Dropbox[1]-pilvipalveluun, josta ne on saatavilla Lively3D:n käyttöön. Lively3D:n työpöydän ja sovellusten tilan voi tallentaa Dropboxiin tai dokumenttitietokantaan Node.js[2]-sovelluksen välityksellä. Lisäksi Lively3D tarvitsee GLGE-kirjastoa 3D-ympäristön piirtämiseen. Lively3D:n osat suhtautuvat toisiinsa kuvassa 5.1 esitetyllä tavalla. [7]

Lively3D:n toteutus pohjautuu vahvasti GLGE-kirjastoon, jota käytetään ympäristön kaiken 3D-sisällön toteuttamiseen. Lively3D:n sovelluslogiikka on myös suunniteltu pitkälti GLGE:n ehdoilla. Lively3D:n sovelluksien ohjelmointirajapinta on määritelty siten, että canvas-sovelluksen kehittäjän ei tarvitse tietää Lively3D:n sisäisistä toteutustekniikoista. 3D-ympäristöjen toteuttaja joutuu puolestaan käyttämään GLGE:tä 3D-ympäristön määrittelemisessä, jotta ympäristö voidaan näyttää oikein Lively3D:ssä. [7]

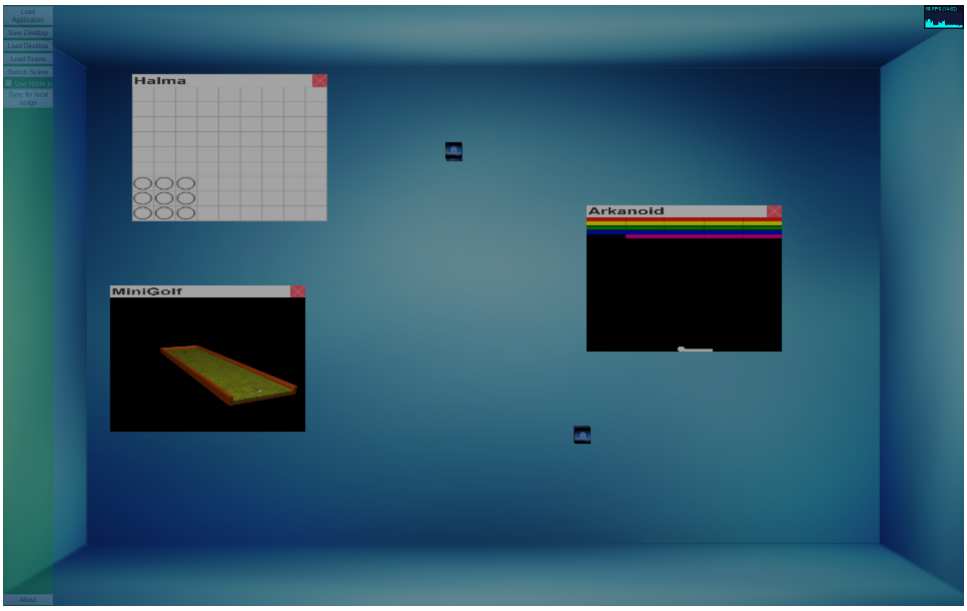
Kuvassa 5.2 on esitetty kuvakaappaus Lively3D:n huoneympäristöstä, jossa on auki kolme sovellusikkunaa ja jossa näkyy myös kahden sovelluksen kuvakkeet. Huoneympäristö toimii Lively3D:n oletusympäristönä. Lively3D:hen on lisäksi toteutet-

[1] Dropbox, https://www.dropbox.com/
[2] Node.js, http://nodejs.org/

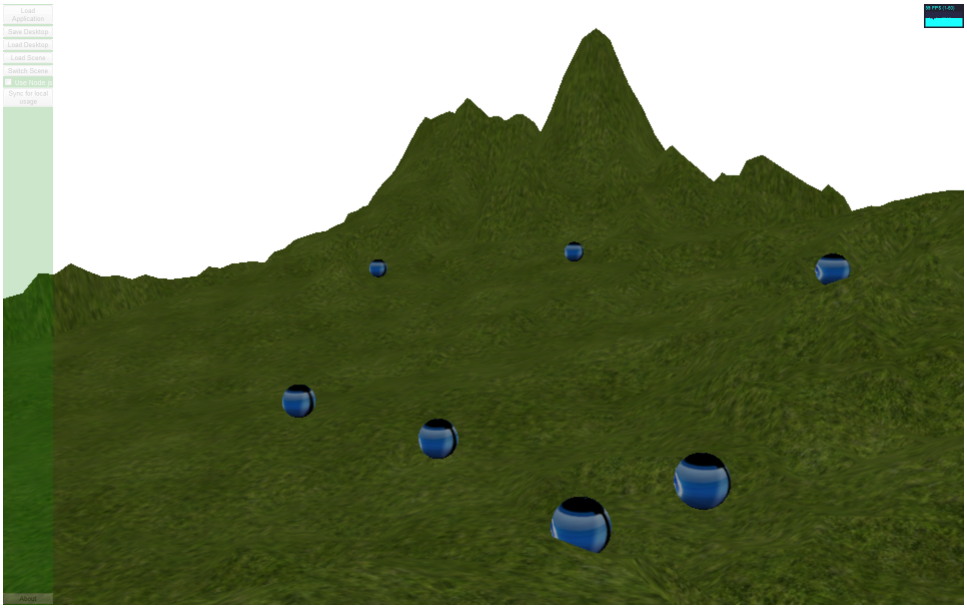
tu myös 3D-maasto- ja aurinkokuntaympäristöt. Kuvassa 5.3 on kuvakaappaus 3D-maastoympäristöstä, jossa näkyvät pallot ovat sovellusten kuvakkeita. Kuvassa 5.4 on esitetty aurinkokuntaympäristö, jossa sovelluskuvakkeet kiertävät aurinkoa.



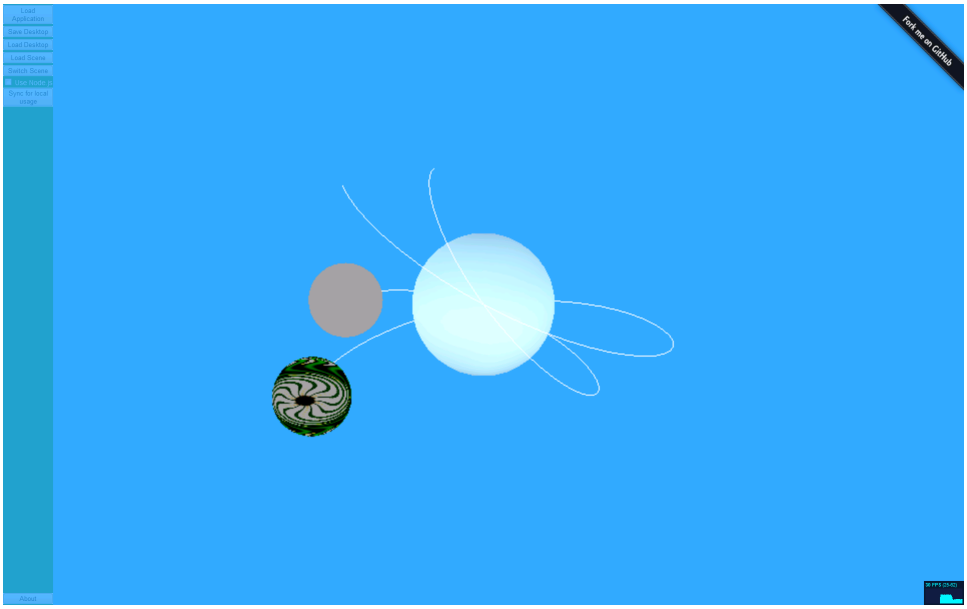
Kuva 5.1: Lively3D:n rakenne [7]



Kuva 5.2: Lively3D:n oletusympäristö, huone



Kuva 5.3: Lively3D:n 3D-maastoympäristö

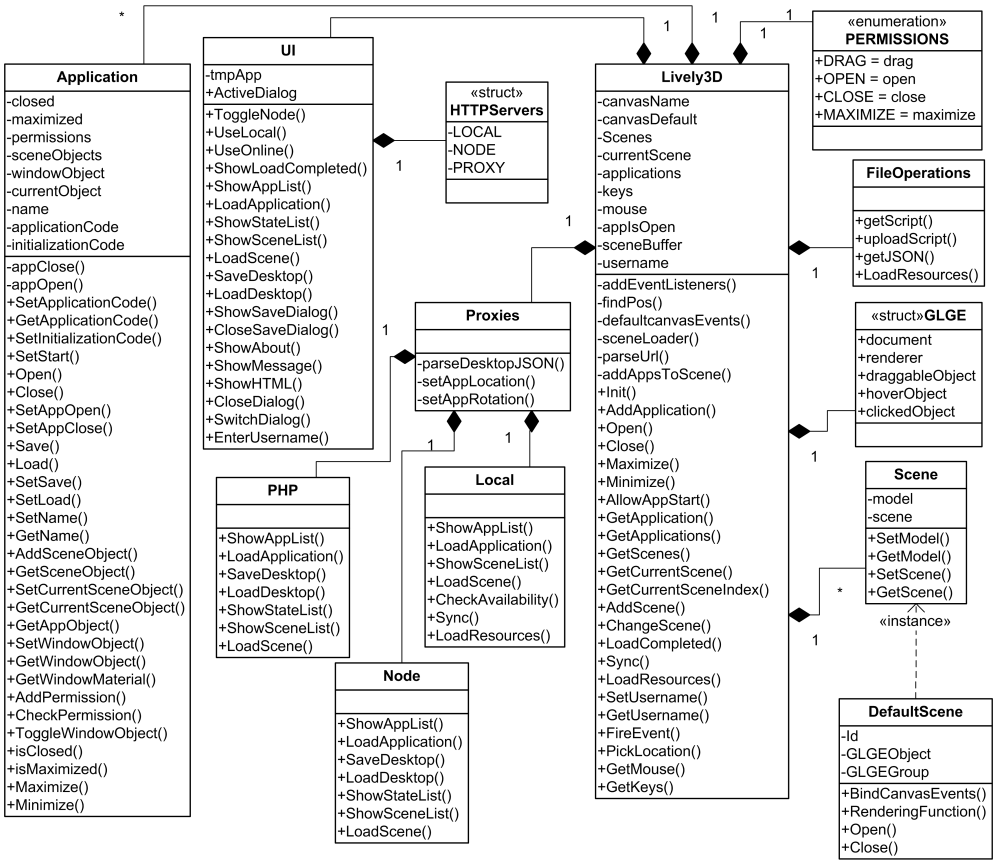


Kuva 5.4: Lively3D:n aurinkokuntaympäristö

5.1.2 Arkkitehtuuri

Kuvassa 5.5 on esitetty Lively3D:n arkkitehtuuri luokkakaaviona. Luokkakaaviossa sovelluslogiikkaa kuvaa luokka *Lively3D*, kaksiulotteista web-käyttöliittymää *UI*, välityspalvelimia *Proxies*, sovellusta *Application* ja 3D-ympäristöä *Scene*. Luokkakaaviossa esitetty *GLGE* on tietue, jossa on sovelluslogiikan tarvitsemia GLGE-kirjastoon liittyviä tietoja. Objekti *DefaultScene* on Lively3D:n oletuksena käyttämä 3D-ympäristö. Luokka *FileOperations* sisältää tiedostojen lukemiseen liittyvät toimenpiteet. Enumeraatio *PER-*

MISSIONS määrittää listan oikeuksista, joita sovellukseen liittyvillä 3D-objekteilla voi olla. Esimerkiksi vain sulkunappia kuvaavalla objektilla on oikeus sulkea sovellus ja sovellusikkunaa voi raahata vain otsikkopalkkiobjektista. 3D-objekteihin liittyviä oikeuksia tarkastellaan tapahtumakäsittelyssä.



Kuva 5.5: Lively3D:n rakenne esitettynä luokkakaaviona

Sovelluslogiikka (Lively3D) sisältää Lively3D:n tietorakenteet ja yleisen hallinta- ja alustuslogiikan. Sovelluslogiikka huolehtii muun muassa tapahtumakäsittelystä, sovellusten lisäämisestä ympäristöön ja visualisointien lisäämisestä ja vaihtamisesta. Sovelluslogiikassa on hyvin paljon GLGE-kirjastosta riippuvia osia, jonka takia se on tämän työn kannalta hyvin tärkeä osa Lively3D-ympäristöä. Uudelleen toteutuksen suurimmat muutokset liittyvät kuitenkin GLGE-kirjaston korvaamiseen. Sovelluksen metodit on kuvattu tarkemmin taulukossa 5.1

Taulukko 5.1: Sovelluslogiikan metodit

Metodi	**Kuvaus**
Init	Sisältää Lively3D:n alustustoimenpiteet.
AddApplication	Lisää sovelluksen ympäristöön.
addAppsToScene	Luo sovelluskuvakkeet.
Open	Kutsutaan, kun sovellus avataan. Kutsuu edelleen 3D-ympäristön ja sovelluksen vastaavaa metodia.
Close	Kutsutaan, kun sovellus suljetaan. Kutsuu edelleen 3D-ympäristön ja sovelluksen vastaavaa metodia.
Maximize	Kutsuu sovelluksen Maximize-metodia.
Minimize	Kutsuu sovelluksen Minimize-metodia.
AllowAppStart	Metodia kutsutaan, kun sovellus on ladattu ja sen voi avata.
GetApplication	Palauttaa sovelluksen, jonka sovellusikkuna vastaa metodille parametrina annettua GLGE:n 3D-objektia.
GetApplications	Palauttaa taulukon ladattuina olevista sovelluksista.
GetScenes	Palauttaa taulukon ladattuina olevista 3D-ympäristöistä.
GetCurrentScene	Palauttaa sillä hetkellä käytössä olevan 3D-ympäristön.
GetCurrentSceneIndex	Palauttaa indeksin jolla käytössä oleva ympäristö on talletettu 3D-ympäristöt sisältävään taulukkoon *scenes*.
AddScene	Lisää 3D-ympäristön Lively3D:hen.
ChangeScene	Vaihtaa käytettävän 3D-ympäristön seuraavaan ladattuna olevaan ympäristöön.
LoadCompleted	kutsuu käyttöliittymän *ShowLoadComplete*-metodia, joka näyttää viestin latauksen valmistumisesta.
Sync	Tallettaa Dropbox-palvelussa olevat resurssit, kuten sovellukset, 3D-ympäristöt ja talletetut tilat, paikalliseen tiedostojärjestelmään *Node*-välityspalvelinta käyttäen.
LoadResources	Lataa sovelluksiin ja 3D-ympäristöihin liittyvät resurssitiedostot, kuten kuvat ja äänet.
sceneLoader	Lataa 3D-ympäristöön liittyvän GLGE:n XML-dokumentin.
parseUrl	Muuntaa 3D-ympäristön GLGE:n XML-dokumentin url:n haluttuun muotoon.
SetUsername	Tallentaa käyttäjän antaman käyttäjänimen.
GetUsername	Palauttaa käyttäjän antaman käyttäjänimen.
defaultCanvasEvents	Lively3D:n tapahtumakäsittelijät.
PickLocation	Tarkastaa, mihin hiiritapahtuma osui 3D-maailmassa.
findPos	Apufunktio hiiritapahtuman törmäystarkastuksen laskemisessa.
FireEvents	Välittää hiiritapahtumat canvas-sovellukselle.
AddEventListeners	Rekisteröi canvas-sovellusten tapahtumakäsittelijät.
GetMouse	Palauttaa GLGE:n hiiritapahtumaobjektin.
GetKeys	Palauttaa GLGE:n näppäintapahtumaobjektin.

Sovellus (Application) sisältää sovellukseen liittyvät 3D-objektit eli sovellusikkunan ja sovelluskuvakkeet jokaista ladattua 3D-ympäristöä kohden. Lisäksi Lively3D-sovellus sisältää rajapinnan, jonka avulla canvas-sovelluksen ohjelmakoodi ja muut tarvittavat tiedot saadaan sidottua Lively3D-sovellukseen. Sovellukseen liittyvät metodit on esitetty taulukoissa 5.2 ja 5.3. Ensimmäisessä taulukossa on esitelty sovellukseen liittyvät Lively3D:n sisäisen toteutuksen käyttämät metodit. Jälkimmäisessä taulukossa on esitelty Lively3D-sovelluksen ohjelmointirajapinnan metodit, joita canvas-sovelluksen tekijä käyttää.

Taulukko 5.2: Sovelluksen Lively3D:n sisäiset metodit

Metodi	**Kuvaus**
Open	Kutsuu appOpen-metodia.
Close	Kutsuu appClose-metodia.
appOpen	Kutsuu sovelluskoodin aloitusmetodia.
appClose	Kutsuu sovelluskoodin sulkumetodia.
Save	Kutsuu sovelluskoodin tilantalletusmetodia.
Load	Kutsuu sovelluskoodin tilanlatausmetodia.
GetName	Palauttaa sovelluksen nimen.
AddSceneObject	Lisää sovellukselle kuvakkeen, joka on GLGE:n 3D-objekti.
GetSceneObject	Palauttaa parametrina annettua 3D-ympäristöä vastaavan sovelluskuvakkeen sisältämät 3D-objektit.
SetCurrentSceneObject	Vaihtaa sovelluksen käytössä olevan kuvakkeen vastaamaan sen hetkisen 3D-ympäristön kuvaketta.
GetCurrentSceneObject	Palauttaa sovelluksen käytössä olevan kuvakkeen.
GetAppObject	Palauttaa parametrina annettua 3D-ympäristöä vastaavan sovelluskuvakkeen.
SetWindowObject	Asettaa sovellukselle ikkunaobjektin, joka on GLGE:n 3D-objekteista koostuva objektijoukko, joka sisältää otsikkopalkin, sulkunapin ja tason, jossa sovellus näytetään.
GetWindowObject	Palauttaa sovellusikkunan.
GetWindowMaterial	Palauttaa sovelluksen canvas-elementin.
AddPermissions	Lisää sovellukseen liittyville 3D-objekteille (kuvakkeet, ikkuna) niiden tarvitsemat oikeudet.
CheckPermissions	Tarkistaa, onko parametrina annetulla GLGE:n 3D-objektilla parametrina annettuja oikeuksia.
ToggleWindowObject	Avaa sovellusikkunan.
isClosed	Kertoo, onko sovellus suljettu.
isMaximized	Kertoo, onko ikkuna suurennettu.
Maximize	Suurentaa sovellusikkunan maksimikokoon.
Minimize	Pienentää sovellusikkunan oletuskokoon.

Taulukko 5.3: Sovelluksen ohjelmointirajapinnan metodit

Metodi	Kuvaus
SetApplicationCode	Sitoo sovellusksen sovelluskoodin, Lively3D:n sovellusobjektiin.
SetInitializationCode	Sitoo sovelluksen alustustoimenpiteet sisältävän funktion sovellusobjektiin.
SetStart	Sitoo funktion, jota kutsutaan, kun sovellus on initialisoitu, sovellusobjektiin.
SetOpen	Sitoo sovelluksen avaamisen toteuttava metodin sovellusobjektiin.
SetClose	Sitoo sovelluksen sulkemisen toteuttavan metodin sovellusobjektiin.
setSave	Sitoo sovelluksen tilan tallettamisen toteuttavan metodin sovellusobjektiin.
setLoad	Sitoo sovelluksen talletetun tilan palauttamisen toteuttavan metodin sovellusobjektiin.
SetName	Asettaa sovelluksen nimen.

Välityspalvelimet (Proxies) huolehtivat Lively3D:n ja palvelimen sekä Dropboxin välisestä kommunikoinnista. Välityspalvelimia on toteutettu kolme: PHP-välityspalvelin, Node.js-välityspalvelin ja lokaali välityspalvelin, joka on tarkoitettu käytettäväksi ilman verkkoyhteyttä. Välityspalvelimet huolehtivat työpöydän tilan talletuksesta sekä talletettujen tilojen, sovellusten ja 3D-ympäristöjen lataamisesta. Välityspalvelimet eivät ole riippuvaisia GLGE-kirjaston käytöstä, joten niitä ei käsitellä tässä tarkemmin.

Käyttöliittymä (UI) määrittelee Lively3D:n kaksiulotteisen web-käyttöliittymän toiminnan. Käyttöliittymä huolehtii dialogien näyttämisen ja vaihtamisen, sekä tiedon vastaanottamisen dialogeista. Käyttöliittymä ei myöskään ole riippuvainen GLGE-kirjastosta vaan se on toteutettu jQuery-kirjaston[3] avulla. Käyttöliittymä esitetään HTML:n avulla ja käyttöliittymän tyyli on määritelty Lively3D:n CSS-tyylitiedostossa.

3D-ympäristö (Scene) määrittelee Lively3D:n visualisointirajapinnan. Kun 3D-ympäristö lisätään, Lively3D luo instanssin *Scene*-luokasta, johon talletetaan 3D-ympäristön malli ja toiminnallisuus. Luokkakaaviossa esitetty *DefaultScene* on Lively3D:n käyttämä oletustoteutus. Attribuutit ja metodit, jotka on kuvattu DefaultScene-objektissa luokkakaaviossa ovat vaadittuja kaikilta 3D-ympäristöiltä. *RenderingFunction*-metodissa määrittellään ympäristöön liittyvät joka piirtosilmukalla suoritettavat toimenpiteet, kuten animointi. *BindCanvasEvents*-metodissa määritetään ympäristön tapahtumakäsittelijät ja metodeissa *Open* ja *Close* määritetään toimenpiteet jotka suoritetaan, kun sovellus avataan tai suljetaan ympäristössä. 3D-ympäristöllä täytyy olla XML-resurssitiedosto, joka sisältää ympäristön määrittelyn GLGE:n tukemassa muodossa.

[3] jQuery, http://jquery.com/

Tiedosto-operaatiot (FileOperations) huolehtivat tiedostoiden lukuun ja kirjoittamiseen liittyvistä toimenpiteistä. Esimerkiksi työpöydän tilan tallettaminen Dropbox-palveluun toimii siten, että tila kirjoitetaan tekstitiedostoon, joka lähetetään palveluun. Kun talletettu tila ladataan, se luetaan tiedostosta.

Kuten aiemmin todettiin, on Lively3D:n toteutus vahvasti riippuvainen GLGE-kirjastosta. GLGE-kirjastoon liittyviä riippuvuuksia löytyy erityisesti sovelluslogiikasta, mutta niitä löytyy myös Lively3D:n sovellusosasta. Visualisointirajapinnassa ei itsessään ole GLGE-riippuvuuksia, mutta Lively3D:n toteutus olettaa, että 3D-ympäristö on toteutettu käyttäen GLGE:tä.

5.1.3 Käyttöliittymä

3D-käyttöliittymän Lively3D:ssä muodostavat sovelluskuvakkeet ja sovellusikkunat. Lively3D:n 3D-käyttöliittymän määrittely ja toiminnallisuuden toteutus on kiinteä osa Lively3D:n sovelluslogiikkaa, mikä voi olla ongelmallista, jos erilaisia käyttöliittymäkomponentteja haluttaisiin lisätä järjestelmään tai esimerkiksi sovellusikkunan ominaisuuksia haluttaisiin muuttaa. Lively3D:n käyttöliittymäkomponentit koostuvatkin oikeastaan pelkästään ulkonäöstä. Komponentteihin ei esimerkiksi voi liittää tapahtumakäsittelijöitä tai muita resursseja.

Sovellusikkunat ja -kuvakkeet ottavat vastaan erilaisia DOM-tapahtumia. Tapahtumien käsittely on toteutettu Lively3D:n sovelluslogiikassa keskitetysti. Hiiritapahtuman tullessa suoritetaan törmäystarkastelu hiiren ja Lively3D:n käyttöliittymään kuuluvien 3D-objektien välillä. Jos tapahtuma osui käyttöliittymässä olevaan 3D-objektiin, suoritetaan toimenpiteet, jotka ovat riippuvaisia siitä, minkä tyyppiseen objektiin tapahtuma kohdistui ja mitä oikeuksia objektilla on. Tapahtumakäsittelijät eivät siis ole käyttöliittymäkomponenttikohtaisia vaan esimerkiksi kaikilla sovelluskuvakkeilla on samat tapahtumakäsittelijät. [7]

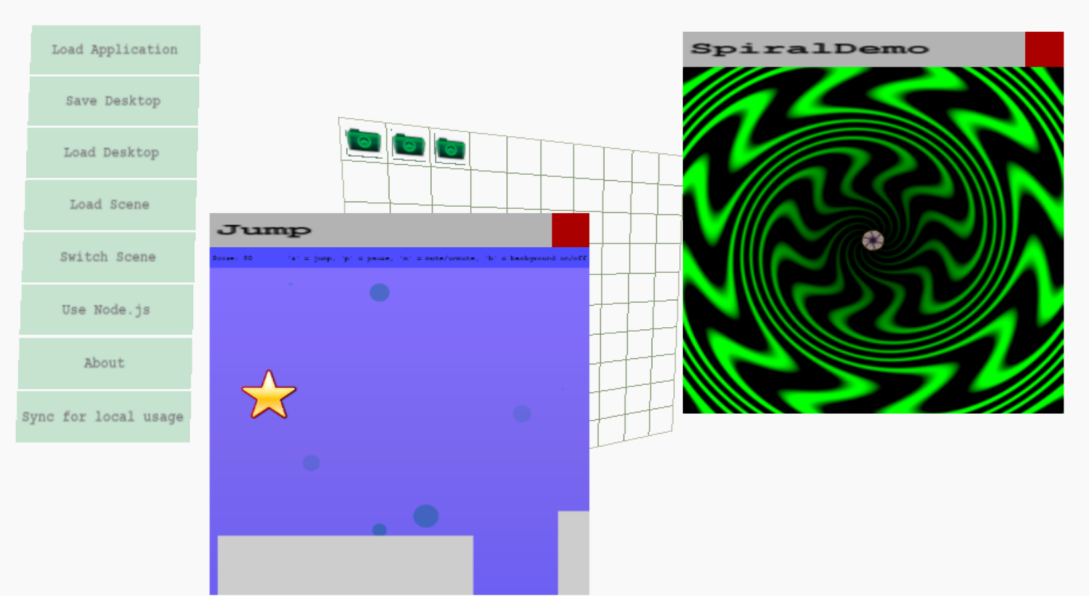
Canvas-sovellukset voivat myös ottaa DOM-tapahtumia vastaan rekisteröimällä tapahtumakäsittelijänsä Lively3D:lle. Näin Lively3D osaa välittää canvas-sovelluksen tapahtuman eteenpäin, jos canvas-sovelluksen ikkuna on aktiivisena. Canvas-sovellukselle välitettävät hiiritapahtumat käsitellään Lively3D:n puolella sellaiseen muotoon, että sovellus saa parametrina tapahtumaobjektin lisäksi tapahtuman x- ja y-koordinaatit canvaksen omassa koordinaatissa. Tämä mahdollistaa sen, että canvas-sovelluksen kehittäjän ei tarvitse tietää oikean klikkauskohdan selvittämiseksi sitä, missä kohdassa ruutua ja minkä kokoisena canvas-sovellus näytetään. [7]

Lively3D:n käyttöliittymä sovellusikkunoita ja sovelluskuvakkeita lukuun ottamatta on toteutettu kaksiulotteisena perinteisenä web-käyttöliittymänä. Käyttöliittymä on toteutettu HTML:n, CSS-tyylien ja jQuery-kirjaston avulla. 2D-käyttöliittymän valikko näkyy edellä esitetyissä kuvissa 5.2, 5.3 ja 5.4 vasemmassa laidassa. Lively3D:n 2D-käyttöliittymä sisältää muutakin kuin kuvissa näkyvän valikkopalkin. Käyttöliittymä si-

sältää dialogit käyttäjätunnuksen kysymiseen ja tilantalletukseen. Lisäksi käyttöliittymä toteuttaa myös sovelluksien, 3D-ympäristöjen ja talletettujen tilojen latausvalikot sekä ympäristöstä kertovan infoviestin ja viestit, jotka näytetään, kun sovellus tai ympäristö on ladattu.

5.2 Lively3D:n muokkaus

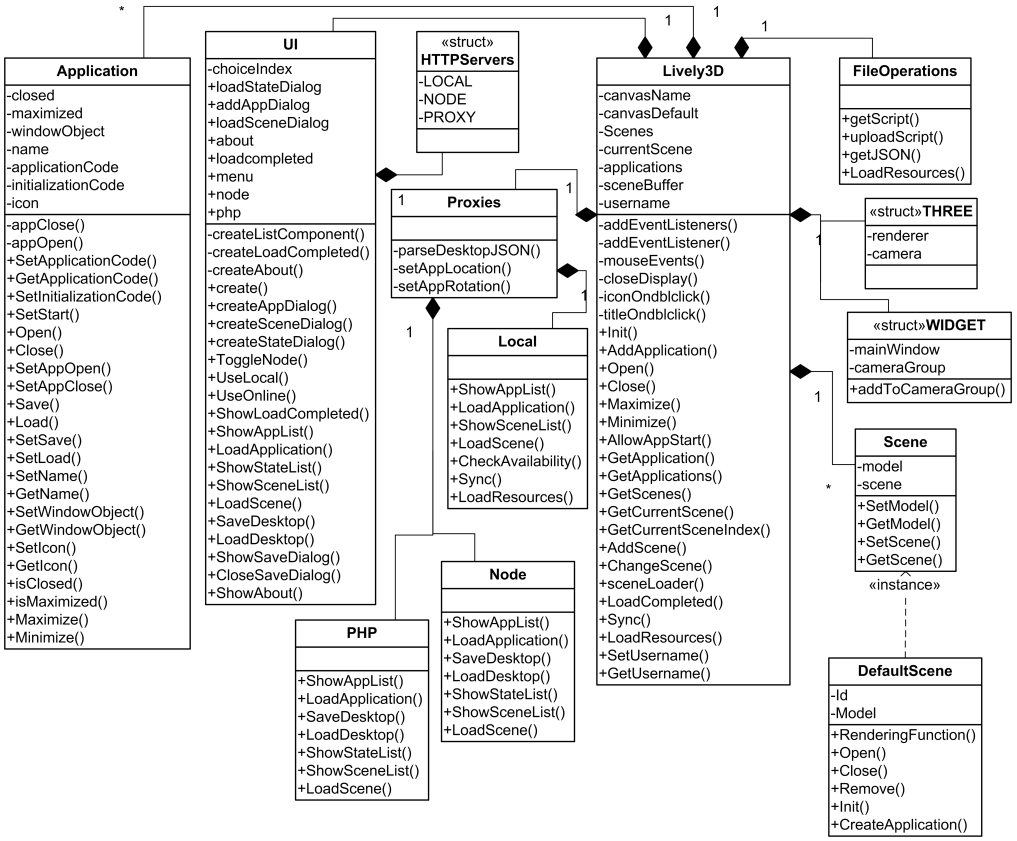
Lively3D muokattiin käyttämään tässä työssä rakennettua käyttöliittymäkomponenttikirjastoa. Lively3D:n alkuperäinen toteutus nojaa vahvasti GLGE 3D-moottoriin, mutta uusi toteutus käyttää three.js 3D-moottoria 3D-ympäristöjen toteuttamiseen ja sovellusten näyttämiseen. Kuvassa 5.6 on kuvakaappaus Lively3D:n uudesta toteutuksesta oletusympäristöllä, jossa on auki kaksi sovellusta ja jossa näkyy kolmen sovelluksen kuvakkeet.



Kuva 5.6: Muokatun Lively3D:n oletusympäristö

5.2.1 Arkkitehtuuri

Suurimmat muutokset Lively3D:n toteutukseen tapahtuivat sovelluslogiikkaan, sovellukseen, käyttöliittymään ja tapaan, jolla 3D-ympäristöt toteutetaan. Kuvassa 5.7 on esitetty kuvaa 5.5 vastaava luokkakaavio uudelleen toteutetun Lively3D:n rakenteesta.



Kuva 5.7: Uudelleentoteutetun Lively3D:n rakenne luokkakaaviona

Muokkauksen myötä Lively3D:n sovelluslogiikka on muuttunut siltä osin, että GLGE-tietue on poistettu ja tilalle on tullut THREE- ja WIDGET-tietueet, jotka sisältävät käyttöliittymäkomponenttikirjastoon ja three.js-kirjastoon liittyviä muuttujia. Sovelluslogiikan metodit ovat muuttuneet jonkin verran. Tapahtumakäsittely toteutetaan nyt käyttöliittymäkomponenttikirjaston tarjoamaa tapahtumajärjestelmää käyttäen. Sovelluksen tapahtumakäsittelijöiden sitomiseksi täytyi tehdä metodit *AddEventListeners* ja *AddEventListener*, jotka suorittavat oikeiden käsittelijöiden sitomisen oikeisiin tapahtumiin. Tapahtumankäsittelyyn liittyvä törmäystarkastelu voitiin poistaa, koska käyttöliittymäkirjaston tapahtumajärjestelmä huolehtii siitä. Myös 3D-objekteille asetettavat oikeudet voitiin poistaa. Sovellusikkunoiden sulkunapille, sovelluskuvakkeille ja sovellusikkunoiden otsikkokentälle on määritelty tapahtumakäsittelijät *closeDisplay*, *iconOndblclick* ja *titleOndblclick*, jotka on sidottu suoraa komponentteihin.

Tapahtuman käsittelyyn jouduttiin toteuttamaa metodi *mouseEvents*, joka vastaa van-

han toteutuksen *FireEvent*-metodia. Hiiritapahtumia ei voida välittää suoraa sovelluksen hiiritapahtumakäsittelijälle, koska sovellusten hiiritapahtumat olettavat saavansa koordinaattitiedot sovelluksen koordinaatistossa, ei Lively3D:n näyttökoordinaatistossa. Tämä aiheuttaa sen, että Lively3D:n sovelluslogiikassa on toteutettava tapahtuman hiirikoordinaattien muunnos sovelluksen koordinaatistoon. Koska komponentit esitetään 3D-maailmassa, pelkän DOM-tapahtumaobjektin mukana tuleva koordinaattitieto ei riitä muunnoksen tekemiseen vaan siihen tarvitaan tieto siitä, mihin kohtaan sovellusikkunan tekstuuria hiiritapahtuma osui. GLGE tarjoaa hiiren törmäyskohdan tesktuurikoordinaatit suoraan, mutta three.js:ssä törmäyskohta on esitetty vain maailmakoordinaatistossa. Three.js-kirjastoa käytettäessä täytyy siis suorittaa hiiren törmäyskohdalle muunnos maailmakoordinaatistosta objektikoordinaatistoon käyttäen three.js:n tarjoamia matriisioperaatioita. Tämän jälkeen voidaan suorittaa muunnos kappaleen tekstuurikoordinaatteihin, jotka voidaan edelleen skaalata sovelluksen koordinaatistoon, kun sovelluksen canvas-elementin koko on tiedossa. Kun sovellukset esitetään kaksiulotteiseen tasoon teksturoituna, on tekstuurikoordinaattien selvittäminen objektikoordinaatistosta yksinkertainen toimenpide, skaalaus välille 0...1. Jos sovellukset esitettäisiin esimerkiksi pallon tai epäsäännöllisen muotoisen kappaleen pinnalle teksturoituna, ei muunnos objektikoordinaatistosta tekstuurikoordinaatistoon ole näin yksinkertainen. Objektin muoto ja se, millä tapaa se on teksturoitu, esimerkiksi onko tekstuuri venytetty kuution päälle vai onko kuution jokainen sivu teksturoitu erikseen, vaikuttavat siihen, miten muunnos objektikoordinaatistosta tekstuurikoordinaatistoon pitää laskea.

Sovelluksien kehittäjälle näkyvä rajapinta on säilytetty ennallaan. Canvas-sovellukset, jotka toimivat alkuperäisessä Lively3D-ympäristössä, toimivat myös uudelleen toteutetussa Lively3D-ympäristössä. Lively3D:n käyttämää osaa sovellusrajapinnasta täytyi kuitenkin muokata, jotta GLGE-riippuvuudet saatiin poistettua. Suurin muutos on se, että sovellus ei enää sisällä kaikkien Lively3D:ssä ladattuna olevien ympäristöjen kuvakkeita vaan se sisältää vain sillä hetkellä käytössä olevan kuvakkeen, joka asetetaan metodilla *setIcon*.

Varsinaiseen 3D-ympäristön rajapintaan ei tehty muutoksia, mutta asiat, joita Lively3D vaatii 3D-ympäristöjen sisäiseltä toteutukselta, muuttuivat paljon. Vanhassa toteutuksessa 3D-ympäristö sisälsi aina XML-tiedoston, joka määritteli ympäristöön kuuluvat 3D-objektit. Three.js 3D-moottorissa 3D-objektien määrittäminen tapahtuu JavaScriptillä. XML-tiedoston sijaan 3D-ympäristön täytyy implementoida alustusfunktio, *Init*, jossa ympäristöön liittyvät mallit luodaan. Alustusfunktiota kutsutaan, kun 3D-ympäristö otetaan käyttöön. Lisäksi ympäristön täytyy toteuttaa metodi *Remove*, joka poistaa ympäristön tietorakenteet ja ympäristöön liittyvät 3D-objektit. Metodia *CreateApplication* käytetään sovelluskuvakkeiden luomiseen. Sovelluskuvakkeet täytyy toteuttaa käyttöliittymäkomponenttikirjaston komponenttina, koska Lively3D:n sovelluslogiikka olettaa niiltä tietyn rajapinnan tapahtumakäsittelyyn. 3D-ympäristön muut 3D-mallit täytyy toteuttaa

joko pelkkää three.js-kirjastoa tai sitä ja käyttöliittymäkomponenttikirjastoa käyttäen.

Käyttöliittymän toiminnot pysyivät samoina, mutta esitystapa muutettiin HTML-käyttöliittymästä 3D-käyttöliittymäksi. Tämän muutoksen takia käyttöliittymään jouduttiin lisäämään joitain metodeja, jotka luovat käyttöliittymän 3D-käyttöliittymäkomponentit.

Alkuperäisessä toteutuksessa osa dialogeista luotiin välityspalvelimissa. Uudessa toteutuksessa dialogien luominen on siirretty käyttöliittymän puolelle. Välityspalvelimiin ei kuitenkaan ole tehty toiminnallisia muutoksia.

5.2.2 Käyttöliittymä

Kuten edellä todettiin, Lively3D:n alkuperäisessä toteutuksessa on vain kahdenlaisia 3D-käyttöliittymäkomponentteja, sovellusikkunoita ja kuvakkeita. Sovellusikkunat on toteutettu uuteen versioon three.js-sovittimen tarjoamilla *TitledWindow*-komponentteilla. Kuvassa 5.2 esitetty oletusympäristö, huone, on korvattu *GridWindow*-komponentilla ja sovelluskuvakkeet esitetään *GridIcon*-komponenteilla. Uusi oletusympäristö näkyy kuvassa 5.6.

Sovellusikkunoiden ja kuvakkeiden toiminta on pidetty samanlaisena alkuperäisen toteutuksen kanssa. Sovellusikkunaa voi siirtää raahaamalla sitä otsikkopalkista ja otsikkopalkin tuplaklikkaaminen muuttaa sovellusikkunan kokoa kuten alkuperäisessä Lively3D:ssä. Kuvakkeen tuplaklikkaaminen avaa sovelluksen.

Käyttöliittymän uudessa toteutuksessa käytettyjen käyttöliittymäkomponenttien esitystapa on määritelty kiinteästi komponenttiin. Esitystapaa ei siis tarvitse enää määritellä sovelluslogiikan joukossa, ja jos esitystapaa halutaan vaihtaa, voidaan komponentti korvata toisella valmiskomponentilla. Käyttöliittymäkomponenttikirjaston tapahtumajärjestelmän avulla käyttöliittymäkomponentteihin voidaan suoraan määrittää käsittelijät tuetuille DOM-tapahtumille. Törmäystarkastelun laskeminen ja oikean komponentin käsittelijän kutsuminen on nyt käyttöliittymäkomponenttikirjaston vastuulla, ei Lively3D:n.

Myös Lively3D:n klassinen kaksiulotteinen web-käyttöliittymä toteutettiin uudelleen 3D-käyttöliittymänä käyttäen three.js-sovittimen valmiita komponentteja. Käyttöliittymäpalkki, joka näkyy vasemmassa reunassa kuvissa 5.2, 5.3 ja 5.4, on korvattu *SelectDialog*-komponentilla, joka voidaan nähdä kuvassa 5.6 vasemmassa laidassa. Myös sovellus-, 3D-ympäristö- ja tilavalikot ovat toteutettu tätä komponenttia käyttäen. Käyttäjätunnuksen kyselyn ja työpöydän tallennuksen dialogit toteutettiin three.js-sovittimen *Dialog*-komponenteilla. Staattiset viestit, kuten lataamisen valmistumisviesti ja infoteksti on toteutettu käyttöliittymäkomponenttikirjaston *Basic*-komponenteille. Komponenttien esitystapa on taso, joka on teksturoitu kuvalla.

Vanhan toteutuksen käyttöliittymä sisälsi metodin *ShowHTML*, jonka avulla suurin osa käyttöliittymän viesteistä näytettiin. Esimerkiksi infoteksti oli toteutettu tätä metodia käyttäen. Teksti on siis määritelty HTML-muodossa ohjelmakoodiin, josta se on muo-

kattavissa melko vaivattomasti verrattuna siihen, mitä uudessa toteutuksessa infotekstin muokkaaminen vaatisi. Infoteksti ja muut Lively3D:n viestit, on uudessa toteutuksena toteutettu kuvana, jota käytetään komponentin tekstuurina. Jos tekstiä halutaan muokata, täytyy muokata tekstuurina käytettävää kuvaa.

2D-käyttöliittymän muuttaminen 3D-käyttöliittymäksi sujui melko mutkattomasti korvaamalla HTML-komponentit 3D-käyttöliittymäkomponenteilla. Ongelmia kuitenkin tuotti se, että 3D-komponentit todellakin ovat osa 3D-maailmaa. Lively3D:n 3D-ympäristöt voivat käyttää kontrolleja, joilla liikutetaan kameraa, näin tekee muun muassa alkuperäisen Lively3D:n maastoympäristö. Kun kameraa liikutetaan, täytyy joidenkin käyttöliittymäkomponenttien, kuten valikoiden, pystyä seuraamaan kameran liikettä, jotta ne olisivat käyttäjän käytettävissä koko ajan. Myös sovellusikkunoiden tulisi aina avautua samaan paikkaan suhteessa kameraan.

Three.js mahdollistaa objektien ryhmittelyn *Object3D*-objektin avulla, jota muun muassa käyttöliittymäkomponenttikirjaston three.js-sovittimessa käytetään ikkunoiden objektijoukkojen toteutuksena. Jos kamera ja kameraa seuraavat käyttöliittymäkomponentit lisää samaan objektijoukkoon ja liikutetaan tätä joukkoa kameran sijaan, saavutetaan haluttu lopputulos. Kuitenkin käyttöliittymäkomponentit kuuluvat automaattisesti johonkin objektijoukkoon ollessaan aina jonkin ikkunan lapsikomponentteja, eli niitä ei voi suoraa lisätä three.js:n objektien lapsiksi rikkomatta käyttöliittymäkomponenttikirjaston sisäistä hierarkiaa. Tämä ongelma ratkaistiin määrittelemällä ikkunakomponentti, *cameraGroup*, jonka lapsiksi kamera ja kaikki kameraa seuraavat käyttöliittymäkomponentit liitetään. Kameran liikuttamisen sijasta, liikutetaan tätä luotua kamerakomponenttia, jolloin kameran lisäksi myös ryhmään kuuluvat käyttöliittymäkomponentit liikkuvat.

Uusien käyttöliittymäkomponenttien lisääminen seuraamaan kameraa sovelluksen ajon aikana on kuitenkin edelleen hieman monimutkaista. Kun kameraan sidottu komponentti asetetaan paikalleen, loogisesti ajatellen sen x-, y- ja z-koordinaattien tulisi kuvata komponentin etäisyyttä kamerasta ennemmin kuin paikkaa 3D-maailmassa. Three.js-kirjastossa komponentit asetetaan kuitenkin paikoilleen käyttäen maailmakoordinaatteja, joten jos kamerakomponentti on liikkunut tai sitä on pyöritetty, täytyy uuden komponentin paikoilleen asettamisessa ottaa tämä huomioon. Ratkaisuksi tähän ongelmaan uuteen Lively3D:hen toteutettiin metodi *addToCameraGroup*, joka tallettaa kamerakomponentin sen hetkiset koordinaatit ja rotaation, siirtää komponentin origoon ja nolla sen rotaatiot. Tämän jälkeen uusi komponentti lisätään kamerakomponentin lapseksi ja kamerakomponentti siirretään takaisin paikalleen ja pyöritetään oikeaan asentoon. Näin kameraa seuraavien komponenttien asettelun yhteydessä ei tarvitse ottaa huomioon kameran sen hetkistä sijaintia vaan komponentin paikka voidaan antaa etäisyytenä kameraan. Kun käyttöliittymä oli toteutettu kaksiulotteisena HTML-tekniikalla, ei edellä kuvattua ongelmaa käyttöliittymäkomponenttien sijoittelussa ollut. HTML-käyttöliittymä piirrettiin aina 3D-maailman päälle ja sen komponenttien paikat ja koot voidaan määritellä näytön koor-

dinaatistossa.

5.3 Arviointi

Lively3D:n käyttöliittymän ja 3D-moottorin korvaaminen onnistui hyvin. Kaikki Lively3D:n toiminnalliset ominaisuudet saatiin säilytettyä ennallaan, ja myös HTML-käyttöliittymä pystyttiin korvaamaan 3D-käyttöliittymällä käyttöliittymäkomponenttikirjastoa käyttäen. Uudelleen toteutuksessa pyrittiin korvaamaan vain välttämättömät osat säilyttäen Lively3D:n arkkitehtuuri mahdollisimman samanlaisena alkuperäisen toteutuksen kanssa, jotta toteutukset olisivat vertailtavissa. Edellä esitettyjä luokkakaavioita (kuvat 5.5 ja 5.7) vertaamalla huomataan, että Lively3D:n arkkitehtuuri ei muuttunut merkittävästi uudelleentoteutuksen myötä. HTML-käyttöliittymää korvattaessa jouduttiin kuitenkin siirtämään ennen välityspalvelimissa toteutettuja dialogien luonteja käyttöliittymän puolelle.

Taulukossa 5.4 on esitetty Lively3D:n uuden ja vanhan toteutuksen JavaScript tiedostojen koodimäärät. Taulukosta nähdään, että koodirivejä uudelleentoteutetun Lively3D:n JavaScript-tiedostoissa on kokonaisuudessaan noin 20% vähemmän kuin vanhassa toteutuksessa. Suurin parannus on sovellusrajapinnan määrittävässä tiedostossa, jonka koodimäärä väheni noin 35% prosentilla. Sovelluslogiikan koodirivimäärä pieneni noin 27%. Sovelluslogiikka sisältää tässä tapauksessa myös tiedosto-operaatiot, koska ne ovat määritelty samaan tiedostoon muun sovelluslogiikan kanssa.

HTML-käyttöliittymän korvaaminen 3D-käyttöliittymällä puolestaan lisäsi koodirivejä käyttöliittymätiedostoon noin 9%. Tässä täytyy ottaa huomioon, että uudessa toteutuksessa on siirretty jonkin verran koodia välityspalvelinten puolelta käyttöliittymään, mikä vaikuttaa varmasti myös tulokseen. Uudelleen toteutetussa versiossa välityspalvelinten rivimäärä on noin 18% pienempi kuin alkuperäisessä toteutuksessa, kuitenkaan toiminnallisia muutoksia ei tehty. Kuten aiemmin todettiin, Lively3D:n visualisointirajapintaa ei muokattu, ja niinpä sen toteutus on sama sekä uudessa että vanhassa versiossa. HTML-käyttöliittymän korvaaminen 3D-käyttöliittymällä vähensi tarvittavan HTML- ja CSS-määrittelyn määrää. CSS-määrittelyn määrä väheni noin 73% ja HTML:n määrä väheni noin 18%.

Taulukko 5.4: Lively3D:n toteutusten väliset erot koodirivimäärässä

Tiedosto	Koodirivejä vanhassa	Koodirivejä uudessa	Muutos
Sovelluslogiikka	725	527	-27%
Sovellus	308	200	-35%
Välityspalvelimet	450	368	-18%
3D-ympäristö	57	57	0%
Käyttöliittymä	269	296	+9%
JavaScriptiä yhteensä	1806	1444	-20%
index.hml	61	50	-18%
lively3d.css	55	15	-73%

2D-käyttöliittymän 3D-käyttöliittymäksi vaihtamisen suurimmiksi ongelmiksi osoittautui dynaamisen tekstin toteuttamisen vaikeus WebGL:llä. Lively3D:n uudesta toteutuksesta poistettiin metodi *ShowHTML* metodilla näytetyt viestit, kuten latauksen onnistuminen ja infoteksti korvattiin kuvilla, joita käytetään käyttöliittymäkomponentin tekstuurina. Komponenttien tekstit eivät näin ollen ole yhtä helposti muokattavissa Lively3D:n uudessa toteutuksessa kuin vanhassa.

Muita ongelmia toteutustekniikan vaihtamisessa tuottivat sovellusten hiiritapahtumien käsittelyyn liittyvät koordinaattimuunnokset ja se, että joidenkin käyttöliittymäkomponenttien tulee pysyä aina samassa kohdassa suhteessa kameraan. Kameraa seuraavat käyttöliittymäkomponenttien toteutus osoittautui lopulta melko suoraviivaiseksi: luodaan komponentti, jonka lapsiksi kamera ja kameraa seuraavat komponentit liitetään, ja liikutetaan tätä komponenttia kameran sijaan.

Canvas-sovellusten vaatimat hiiritapahtumien koordinaattimuunnokset ovat edellistä haastavampi ongelma. Lively3D:n tapauksessa koordinaattimuunnosten laskeminen on suoraviivaista, vaikkakin hieman työlästä. Edellisessä kohdassa kuvattu ratkaisu, joka tällä hetkellä Lively3D:hen on toteutettu, toimii ainoastaan silloin, kun sovellukset on esitetty tason pinnalle teksturoituna. Tämä pitää ottaa huomioon, jos sovellusikkunoiden esitystapaa halutaan myöhemmin muuttaa.

Lively3D:n toteutusten vertailusta voidaan päätellä, että rakennetun käyttöliittymäkomponenttikirjaston käyttö on vähentänyt kokonaisuudessaan sovelluskehittäjän kirjoittaman sovelluskoodin määrää huomattavasti. Lively3D:n muokkaus osoittaa myös sen, että kaksiulotteinen web-käyttöliittymä on myös mahdollista korvata 3D-käyttöliittymällä rakennettua komponenttikirjastoa käyttäen.

6 YHTEENVETO

Työssä rakennettu käyttöliittymäkomponenttikirjasto helpottaa kolmiulotteisten web-käyttöliittymien toteuttamista nostamalla abstraktiotason 3D-moottorin primitiiveistä 3D-käyttöliittymäkomponentteihin. Kirjasto tarjoaa käyttöliittymäkomponenteille yhtenäisen rajapinnan niiden liikuttamiseen, näkyvyyden muuttamiseen ja hiiri- ja näppäimistötapahtumien vastaanottamiseen. Kirjasto abstrahoi käyttäjältä tapahtumankäsittelyn sekä käyttöliittymäkomponenttien esitystavan määrittämisen. Kirjasto mahdollistaa myös valmiiden komponenttien laajentamisen.

Kirjasto käyttää 3D-moottoria sovittimen kautta, mikä mahdollistaa kirjaston käyttämisen eri 3D-moottoreilla ja helpottaa 3D-moottorin vaihtamista. Se, että kirjaston runko ei ole riippuvainen suoraan 3D-moottorista mahdollistaa kirjaston käytön myös muiden kuin WebGL-pohjaisten 3D-moottorien kanssa. Kirjastoon voi halutessaan toteuttaa sovittimen esimerkiksi canvas-, SVG- tai CSS-3D-moottoreille. Kuitenkin kirjaston tapahtumajärjestelmästä on eniten hyötyä WebGL- ja canvas-moottoreiden kanssa käytettynä. Näissä tekniikoissa interaktiiviset komponentit esitetään canvas-elementin sisällä eikä niihin näin ollen ole mahdollista liittää tapahtumakäsittelijöitä suoraan selaimen ohjelmointirajapinnan kautta.

Sovittimia työn puitteissa tehtiin vain yhteen 3D-moottoriin. Tällä hetkellä kirjaston käyttäminen toisella 3D-moottorilla vaatisi uuden sovittimen kirjoittamista. Kirjasto tällaisenaan ei siis vielä toteuta tavoitettaan yleiskäyttöisenä 3D-moottorista riippumattomana käyttöliittymäkomponenttikirjastona.

Kirjasto tarjoaa tällä hetkellä vain suppean määrän komponentteja verrattuna vaikkapa wxWidgets-kirjaston tarjoamaan komponenttien määrään. Sovellusohjelmoija joutuu näin ollen laajentamaan valmiita komponentteja. Sovittimen tarjoamat valmiskomponentit helpottavat käyttäjää, koska niiden toteutus abstrahoi komponentin esitystavan määrittämisen käyttäjältä. Valmiskomponentteja toteutettiin työn puitteissa kuitenkin vain muutama. Toteutettu käyttöliittymäkomponenttikirjasto ei näin ollen riitä abstrahoimaan 3D-moottorin käyttöä täysin, vaan kirjastoa käyttävän sovelluskehittäjän täytyy osata käyttää myös 3D-moottoria saadakseen kirjastosta mahdollisimman paljon hyötyä.

Sovittimen valmiskomponentit muistuttavat esitystavaltaan perinteisiä kaksiulotteisia käyttöliittymäkomponentteja: toteutettujen ikkunakomponenttien muoto on kaksiulotteinen taso kolmiulotteisessa maailmassa. Toisaalta 3D-käyttöliittymäkomponentit mahdollistavat paljon mielikuvituksellisempiakin esitystapoja. On täysin mahdollista toteuttaa

esimerkiksi hakemistoikkunakomponentti, kuten *GridWindow*-komponentti, jonka esitystapana on puun 3D-malli ja jossa hakemistossa olevat tiedostot kuvataan puun lehtinä.

Tällä hetkellä toteutettu kirjasto ei tarjoa käyttöliittymän toteuttamista helpottavia näkymättömiä komponentteja, kuten esimerkiksi komponenttien asettelusta huolehtivia pohjapiirustusluokkia. Kirjasto ei myöskään sisällä mitään yleispäteviä käyttöliittymäkirjastojen palveluja, kuten käyttäjän syötteen validointia. Toteutettua käyttöliittymäkomponenttikirjasto on kuitenkin mahdollista laajentaa tarjoamaan tällaisia palveluita.

Komponenttikirjastoa toteutettaessa ongelmia tuotti WebGL 3D-moottorien puutteellinen dokumentointi. Monen kirjaston dokumentointi nojaa pelkästään esimerkkisovelluksiin, joissa kirjaston käyttöä ei välttämättä esitellä kovinkaan laajasti. WebGL 3D-moottorit kehittyvät myös nopeasti. Pahimmissa tapauksissa työssä käytetyn 3D-moottorin version päivittäminen vaati koko 3D-moottorista riippuvan sovittimen uudelleen toteuttamista. Tämän takia työn toteutuksessa riippumattomuus 3D-moottorista on ollut merkittävä suunnittelulähtökohta. Kirjaston runko-osan 3D-moottorista riippumattomuuden ansiosta kirjastoa voitiin kehittää eteenpäin siten, että ohjelmakoodiin täytyi tehdä mahdollisimman vähän muutoksia 3D-moottorin päivittyessä.

Lively3D-ympäristön käyttöliittymän uudelleentoteutus komponenttikirjaston avulla osoittaa sen, että komponenttikirjaston käyttö yksinkertaistaa 3D-käyttöliittymien toteuttamista. Lively3D:n kaksiulotteisen käyttöliittymän korvaaminen 3D-käyttöliittymällä todettiin myös mahdolliseksi. Sovelluskoodia Lively3D:n uudessa toteutuksessa on noin 20% vähemmän kuin vanhassa toteutuksessa. Myös CSS-määrittelyn määrä väheni noin 75% HTML-käyttöliittymän korvaamisen myötä.

3D-käyttöliittymien toteuttaminen komponenttikirjastoa käyttäen on kuitenkin vielä jonkin verran monimutkaisempaa kuin esimerkiksi HTML-käyttöliittymän tekeminen. Vaikka kokonaisuudessaan käyttöliittymäkomponenttikirjaston käyttö vähensi Lively3D:n sovelluskoodin määrää, 3D-käyttöliittymän toteutus sisälsi noin 9% enemmän rivejä kuin vastaavan HTML-käyttöliittymän toteutus. HTML-käyttöliittymän korvaamisessa 3D-käyttöliittymällä täytyy muistaa ottaa myös huomioon se, että 3D-käyttöliittymä on osa 3D-maailmaa. Jos käyttöliittymäkomponenttien tulee olla jatkuvasti käyttäjän näkyvissä, ne täytyy muistaa sitoa seuraamaan kameraa, jottei käyttäjä eksyisi niistä liikkuessaan 3D-maailmassa.

Dynaamisen tekstin toteutus WebGL:llä ei ole yksinkertaista ja myöskään kaikki 3D-moottorit eivät tarjoa palveluita dynaamisen tekstin toteuttamiseen. Helpoin tapa toteuttaa dynaamista tekstiä WebGL-pohjaisilla järjestelmillä on käyttää tekstuurina canvas-elementtiä, jolle teksti piirretään. Dynaamisen tekstin toteuttamisen vaikeus vaikeuttaa myös geneeristen 3D-käyttöliittymien toteuttamista.

Yhteenvetona voidaan todeta, että web kehitysalustana tarjoaa mahdollisuuden toteuttaa interaktiivisia sovelluksia, jotka eivät ole riippuvaisia käyttöjärjestelmästä. Uusien standardien myötä myös ilman liitännäisiä toteutettujen graafisesti intensiivis-

ten 3D-sovellusten kehittämisestä on tullut mahdollista. WebGL:n päälle on rakennettu lukuisia apukirjastoja ja 3D-moottoreita, jotka helpottavat 3D-sovellusten kehitystä entisestään. Tässä työssä esitelty lähestymistapa nostaa kolmiulotteisten vuorovaikutteisten web-sovellusten toteuttamisen abstraktiotasoa esittelemällä tavan toteuttaa 3D-käyttöliittymäkomponentteja. Työssä toteutettu komponenttikirjasto helpottaa 3D-käyttöliittymien toteuttamista, vaikka se jääkin ominaisuuksiltaan vielä kauas perinteisten työpöytäsovellusten toteuttamiseen käytettävistä käyttöliittymäkirjastoista.

LÄHTEET

[1] A. Taivalsaari, T. Mikkonen, M. Anttonen, and A. Salminen. The Death of Binary Software: End User Software Moves to the Web. *2011 Ninth International Conference on Creating, Connecting and Collaborating through Computing*, pages 17–23, 2011.

[2] A. Taivalsaari. Mashware: the Future of Web Applications. Technical Report TR-2009-181, Sun Microsystem Laboratories, Helmikuu 2009.

[3] I. Hickson. HTML5, A vocabulary and associated APIs for HTML and XHTML. Technical report, W3C, 2011. http://www.w3.org/TR/html5/spec.html.

[4] C. Marrin. WebGL Specification. Technical report, Khronos Group, 2011. http://www.khronos.org/registry/webgl/specs/1.0/.

[5] D.A. Bowman, A. Kruijff, and J.J. LaViola Jr. *3D User Interfaces: Theory and Practice*. Addison Wesley, 2005.

[6] J.M. Gozálex-Calleros, J. Vanderdonckt, and J. Muñoz-Arteaga. A Structured Approach to Support 3D User Interface Development. *2009 Second International Conferences on Advances in Computer-Human interactions*, pages 75–81, 2009.

[7] J-P. Voutilainen. Vuorovaikutteisten web-sovellusten kehittäminen, Joulukuu 2011. Diplomityö, Tampereen teknillinen yliopisto, http://URN.fi/URN:NBN:fi:tty-2011122014956.

[8] L. Kluger. The Open Look Graphical User Interface and its Toolkits. *User Interface Management Systems, IEE Colloquium*, marraskuu 1989.

[9] H. Oldenburg. OSF Motif The User Interface Standard. *User Interface Management Systems, IEE Colloquium*, marraskuu 1989.

[10] S. Hansen and T.V. Fossum. *Event Based Programming*. 2010. http://ginger.cs.uwp.edu/staff/hansen/EventsWWW/Text/Events.pdf.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1996.

[12] wxWidgets, viralliset kotisivut. http://www.wxwidgets.org/about/, verkkosivu, viitattu 10.5.2012.

[13] J. Smart. R. Roebling. V. Zeitlin. R. Dunn. et al. WxWidgets 2.8.12: A Portable C++ and Python GUI Toolkit. Technical report, 2011. http://docs.wxwidgets.org/stable/wx_contents.html.

[14] A. Puhakka. *3D-grafiikka*. Talentum Media, 2008.

[15] C.Marrin, D. Jackson, and D. Hyatt. CSS 3D Transforms Module Level 3. Technical report, W3C, 2009. http://www.w3.org/TR/css3-3d-transforms/.

[16] E. Dahlström, P. Dengler, A. Grasso, C. Lilley, C. McCormack, D. Schepers, J. Ferraiolo, F. Jun, and D. Jackson. Scalable Vector Graphics (SVG) 1.1 (Second Edition). Technical report, W3C, 2011. http://www.w3.org/TR/SVG11/.

[17] SVG3D - Adding a third dimension to svg pictures. http://code.google.com/p/svg3d/, verkkosivu, viitattu 22.3.2012.

[18] A. Munshi and J. Leech. OpenGL ES Common Profile Specification. Technical report, Khronos Group, marraskuu 2010. http://www.khronos.org/registry/gles/specs/2.0/es_full_spec_2.0.25.pdf/.

[19] Blacklists And Whitelists. http://www.khronos.org/webgl/wiki/BlacklistsAndWhitelists, verkkosivu, viitattu 7.5.2012.

[20] Introducing the ANGLE Project. http://blog.chromium.org/2010/03/introducing-angle-project.html, verkkosivu, viitattu 8.5.2012.

[21] WebGL Considered Harmful. http://blogs.technet.com/b/srd/archive/2011/06/16/webgl-considered-harmful.aspx, verkkosivu, viitattu 8.5.2012.

[22] SwiftShader - Software 3D Rendering. http://updates.html5rocks.com/2012/02/SwiftShader-brings-software-3D-rendering-to-Chrome, verkkosivu, viitattu 7.5.2012.

[23] Three.js, github repositorio. http://github.com/mrdoob/three.js, verkkosivu, viitattu 29.8.2012.

[24] M. Anttonen and A. Salminen. Building 3D WebGL Applications. Technical report, Tampereen teknillinen yliopisto, 2011. http://URN.fi/URN:NBN:fi:tty-2011101014827.

[25] GLGE, github repositorio. http://github.com/supereggbert/GLGE, verkkosivu, viitattu 29.8.2012.

[26] SceneJS, github repositorio. http://github.com/xeolabs/scenejs, verkkosivu, viitattu 29.8.2012.

A YKSINKERTAINEN KUVASELAIN TOTEUTETTUNA THREE.JS 3D-MOOTTORILLA

Ohjelma A.1: Yksinkertaisen kuvaselaimen toteutus trhee.js 3D-moottoria käyttäen.

```
1  //Funktio, jota kutsutaan, kun HTML-dokumentti on latautunut.
2  var kuvaselain = function(){
3
4    //Taulukko kuvista, jotka esitetään kuvaselaimessa.
5    var pictures =
6      ["img/kitten1.jpg", "img/bunny1.jpg", "img/rat1.jpg", "img/bunny2.jpg"];
7
8    //Luodaan three.js:n renderer objekti, joka hoitaa piirtämisen ja webGL:n alustamisen.
9    var renderer = new THREE.WebGLRenderer();
10   //Asetetaan canvas-elementille koko, tässä asetetaan canvas selainikkunan kokoiseksi.
11   renderer.setSize( window.innerWidth, window.innerHeight );
12   //Asetetaan ruudun puhdistusväriksi vaalean harmaa.
13   renderer.setClearColorHex( 0xf9f9f9, 1 );
14   //Lisätään canvaselementti HTML-dokumenttiin.
15   document.body.appendChild( renderer.domElement );
16
17   //Luodaan three.js perspektiivikamera.
18   var camera = new THREE.PerspectiveCamera(75, window.innerWidth / window.innerHeight,
         1, 10000);
19   //siirretään kamera haluttuun paikkaan.
20   camera.position.z = 1600;
21
22   //Luodaan 3D-maalima, scene.
23   var scene = new THREE.Scene();
24
25   //Three.js:n projektorin avulla voidana laskea muunnoksia esimerkiksi
         maailmakoordinaatiston ja objektikoordinaatiston välillä ja tarkastaa
         hiiritapahtumien ja objektien välisiä törmäyksiä.
26   var projector = new THREE.Projector();
27
28   //Luodaan 3D-objektijoukko, jolla kuvat ja ruudukko liitetään yhteen.
29   var container = new THREE.Object3D();
30
31   //Lisätään objektijoukko 3D-maailmaan. Nyt kaikki objektijoukkoon lisättävät 3D-
         objektit lisätään automaattisesti 3D-maailmaan.
32   scene.add(container);
33
34   var gridWidth = 2000;
35   var gridHeight = 2000;
36   var gridDensity = 10;
37   //Luodaan ruudukon geometria. Ruudukko on 2000 x 2000 taso, joka koostuu 10*10
         ruudukosta.
38   var gridGeometry = new THREE.PlaneGeometry( gridWidth, gridHeight, gridDensity,
         gridDensity );
39   //Luodaan ruudukkotason materiaali.
```

```
40    var gridMaterial = new THREE.MeshBasicMaterial( { color: 0xFF90BF, opacity: 0.5,
          wireframe: true, wireframeLinewidth : 2} );
41    //Luodaan ruudukon 3D-objekti.
42    var gridMesh =  new THREE.Mesh( gridGeometry, gridMaterial );
43    //Asetetaan taso sellaiseksi, että se on näkyvä kummaltakin puolelta.
44    gridMesh.doubleSided = true;
45    gridMesh.flipSided = true;
46    //Käännetään ruudukko xz-tasosta xy-tasoon.
47    gridMesh.rotation.x = Math.PI/2;
48
49    //Lisätään ruudukko objektijoukkoon.
50    container.add(gridMesh);
51
52    //Apumuuttujat kuvien asettelua varten.
53    var gridLeft = - gridWidth/2.0 + gridMesh.position.x/gridWidth;
54    var gridTop =  gridHeight/2.0 + gridMesh.position.y/gridHeight;
55
56    var stepX = gridWidth/gridDensity;
57    var stepY = gridHeight/gridDensity;
58
59    var slotCenterX = stepX/2;
60    var slotCenterY = stepY/2;
61
62    var x;
63    var y;
64
65    //Luodaan kuvat ja asetetaan ne paikalleen ruudukkoon.
66    for (var i = 0; i < pictures.length; ++i){
67      //Luodaan kuutio, jonka leveys on vähän pienempi ruudukon yhden ruudun leveyttä ja
            korkeus hieman pienempi yhden ruudun korkeutta. Kuution syvyys on 30.
68      var geometry = new THREE.CubeGeometry((gridWidth/(gridDensity+3.3)), (gridHeight/(
            gridDensity+3.3)), 30);
69      //Luodaan three.js tekstuuri ja materiaali.
70      var texture = THREE.ImageUtils.loadTexture(pictures[i]);
71      var material = new THREE.MeshBasicMaterial({map : texture});
72      //Luodaan 3D-objekti.
73      var mesh = new THREE.Mesh(geometry, material);
74
75      //Asetetaan kuva paikalleen ruudukkoon.
76      if(i > 0){
77
78        if(i % gridDensity == 0)
79        {
80          x = gridLeft + slotCenterX;
81          y = y - stepY;
82        }
83        else{
84
85          x = x + stepX;
86          y = y;
87
88        }
89      }
90      else{
91        x = gridLeft + slotCenterX;
92        y = gridTop - slotCenterY;
93      }
94      mesh.position.x = x;
95      mesh.position.y = y;
```

```
96      //Lisätään objekti objektijoukkoo.
97      container.add(mesh);
98    }
99
100   //Määritellään tapahtumakäsittelijät ja apumuuttujat ruudukon pyörittämistä varten.
101   var clickLocation;
102   var rotationOnMouseDownY;
103   var rotationOnMousedownX;
104   var modelRotationY = 0;
105   var modelRotationX = 0;
106   var rotate = false;
107
108
109   var onmousedown = function(event){
110
111     clickLocation = calculateMousePosition(event);
112
113     //Tarkastetaan, osuiko hiiritapahtuma ruudukkoon.
114     if(collisionDetection(clickLocation) == gridMesh){
115       rotate = true;
116       rotationOnMouseDownY = modelRotationY;
117       rotationOnMouseDownX = modelRotationX;
118     }
119   }
120
121   var onmousemove = function(event){
122
123     var mouse = calculateMousePosition(event)
124
125     //Lasketaan, paljonko hiiri on liikkunu, kun nappi on painettu pohjaan. Tämä määrää,
          kuinka paljon ruudukkoa pyöritetään.
126     if (rotate && collisionDetection(mouse) == gridMesh){
127       modelRotationY = rotationOnMouseDownY + ( mouse.x - clickLocation.x );
128       modelRotationX = rotationOnMouseDownX + ( mouse.y - clickLocation.y );
129     }
130   }
131
132   var onmouseup = function(event){
133     //Kun hiiren nappi nousee, ruudukkoa ei enää pyöritetä.
134     rotate = false;
135   }
136
137   //Rekisteröidään tapahtumakäsittelijät canvas-elementille.
138   renderer.domElement.onmousedown = onmousedown;
139   renderer.domElement.onmousemove = onmousemove;
140   renderer.domElement.onmouseup   = onmouseup;
141
142   //Apufunktio hiiren koordinaattien selvittämiseen.
143   var calculateMousePosition = function(event){
144     //Muunetaan hiirikoordinaatit näytön koordinaatistosta canvas-elementin
          koordinaatistoon.
145     var mouseCoordinates = { x: 0, y: 0};
146     if (!event) {
147       event = window.event;
148       mouseCoordinates.x = event.x;
149       mouseCoordinates.y = event.y;
150     }
151     else {
152       var element = event.target ;
```

```
153         var totalOffsetLeft = 0;
154         var totalOffsetTop = 0 ;
155
156         while (element.offsetParent)
157         {
158           totalOffsetLeft += element.offsetLeft;
159           totalOffsetTop += element.offsetTop;
160           element = element.offsetParent;
161         }
162         mouseCoordinates.x = event.pageX - totalOffsetLeft;
163         mouseCoordinates.y = event.pageY - totalOffsetTop;
164       }
165       //Skaalataan koordinaatit välille -1..1
166       var x = +(mouseCoordinates.x / renderer.domElement.width) * 2 - 1;
167       var y = -(mouseCoordinates.y / renderer.domElement.height) * 2 + 1;
168
169       return {x : x, y : y};
170     }
171
172     //Törmäystarkastelufunktiossa tarkastetaan, osuuko hiiritapahtuma ruudukkoon.
173     var collisionDetection = function(mouse){
174       //talletetaan hiirikoordinaatit three.js-vektoriin.
175       var vector  = new THREE.Vector3(mouse.x, mouse.y, 1);
176       //Luodaan poimintasäde.
177       var ray = projector.pickingRay(vector, camera);
178       //Tarkastetaan osuuko säde ruudukkoon.
179       return ray.intersectObject(gridMesh);
180     }
181
182     //Animointifunktio, jossa pyöritetetään objektijoukkoa, johon ruudukko kuuluu. Näin
          myös kuvat pöyrivät saman keskipisteen ympäri saman verran.
183     var animate = function(){
184       container.rotation.y = container.rotation.y + ((modelRotationY - container.rotation.
            y)*0.03);
185       container.rotation.x = container.rotation.x + ((modelRotationX - container.rotation.
            x)*0.03);
186     }
187
188     //Animointisilmukan toteutus.
189     var mainLoop = function(){
190       //Suoritetaan animointi.
191       animate();
192       //Piirretään maailma
193       renderer.render(camera, scene);
194     };
195
196     //Asetetaan intervalli (millisekuntteina), jonka välein animointisilmukkaa kutsutaan.
          Nyt funktiota kutsutaan 60 kertaa sekunnissa eli maksimi FPS on 60.
197     setInterval(mainLoop, 1000/60);
198   }
```